Universidad Carlos III de Madrid
Escuela Politécnica Superior
Grado en Ingeniería Informática

Trabajo de Fin de Grado

Evaluación del Rendimiento de Aplicaciones basadas en Patrones Paralelos

Author: Laura Martín Gallardo

Supervisor: José Daniel García Sánchez

Leganés, Madrid, España
Junio 2018

Agradecimientos

En primer lugar, quisiera dar las gracias a mi tutor por haber confiado en mí y haberme dado la oportunidad de realizar este proyecto.

A Manuel Francisco Dolz, David del Río y Javier Fernández por su paciencia y apoyo, que siempre que se me presentaba una dificultad, ahí estaban, dedicándome parte de su tiempo para ayudarme y asesorarme.

A mis compañeros de laboratorio y amigos, José Cabrero, Nerea Luna y Pablo Brox por aguantarme cada día e intentar echarme una mano siempre que me desesperaba porque algo no me salía.

Y sobre todo a mis padres, mi hermano y mi familia, porque son una fuente de inspiración para mí y me han ayudado y animado en todo momento a no darme por vencida.

Evaluación del Rendimiento de Aplicaciones basadas en Patrones Paralelos

Trabajo Fin de Grado

Laura Martín Gallardo

Resumen

El incremento de los últimos años de la carga de trabajo de los programas ha supuesto el desarrollo de nuevas técnicas para la mejora del rendimiento, tales como las arquitecturas multinúcleo y computación paralela. Este tipo de arquitecturas tiene una mayor capacidad de procesamiento ya que consta de más unidades de cómputo, lo que favorece su uso en conjunto con la programación paralela.

Este tipo de computación permite la ejecución simultánea de varias porciones del código, disminuyendo de esta forma el tiempo de ejecución necesario. Ésta se puede implementar por medio de APIs, modelos de programación, bibliotecas, etc, aunque este trabajo de fin de grado se ha centrado en los modelos de programación. Existen una gran variedad estos modelos, aunque se han seleccionado solo algunos de los disponibles para el lenguaje de programación C++. Los modelos seleccionados se han clasificado en: modelos de alto nivel, modelos de bajo nivel y modelos basados en patrones paralelos.

El objetivo de este trabajo de fin de grado es evaluar todos esos modelos de programación paralela que se han seleccionado usando para ello aplicaciones paralelizables. Para seleccionar dichas aplicaciones fue necesario analizar distintos *benchmarks*, finalmente se seleccionó el *benchmark* P^3ARSEC pues permitía evaluar modelos de programación de los tres tipos propuestos.

Antes de pasar directamente a la evaluación se decidió incorporar una nueva versión paralela a cada una de las aplicaciones, la versión implementada con el modelo de programación basado en patrones paralelos *GrPPI*, puesto que éste tiene un diseño ideal para evaluar distintos modelos al integrar internamente implementaciones de estos.

Como se demuestra en la evaluación, no hay un modelo que sea mejor que otro, sino que depende del contexto en el que se use, por ello es necesario encontrar el modelo adecuado para cada aplicación.

Palabras clave: Modelos de programación · Computación Paralela · Patrones Paralelos · Rendimiento ·P^3ARSEC

Performance Evaluation of Parallel Patterns Applications

Bachelor's Thesis

Laura Martín Gallardo

Abstract

In recent years, there had been an increase of programs workload which has led to the development of new techniques for improving performance, such as multicore architectures and parallel computing. This type of architectures has a greater processing capacity since it is composed of several computing units, which favours its use hand in hand with parallel programming.

This type of programming allows the simultaneous execution of several portions of the code, reducing like this the necessary execution time. This can be implemented through APIs, programming models, libraries, etc, although this bachelor's thesis has focused on programming models. There are many different models even if only some of them have been selected. The selected models have been classified into high-level models, low-level models and models based on parallel patterns.

This bachelor's thesis aim consists of evaluating all those parallel programming models that have been selected using parallel applications. In order to select these applications, it was necessary to analyse different benchmarks however, the one selected was PARSEC benchmark as it allowed evaluating the three types of programming models.

Instead of going directly to the evaluation, it was implemented a new version using the programming model based on parallel patterns GrPPI, since it has an ideal design to evaluate different models considering that it internally integrates its implementations.

As shown in the evaluation section, there is no model that can be considered as the best but depends on the context in which it is used, so it is necessary to look for the proper model for each application.

Keywords: Programming Models · Parallel Programming · Parallel Patterns · Performance · PARSEC

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En los últimos años ha ido surgiendo la necesidad de resolver problemas cada vez más complejos, es decir, problemas que requieren un tiempo de ejecución bastante más elevado que los que se habían estado realizando hasta ese momento. Ejecutar este tipo de programas de forma secuencial en ordenadores con un único núcleo conlleva una cantidad de tiempo demasiado elevada, lo que ha dado lugar a un aumento del interés general en el desarrollo y mejora de técnicas alternativas capaces de aumentar la eficiencia y el rendimiento, reduciendo así el tiempo de ejecución de estos programas. Algunas de estas técnicas o herramientas que se han ido desarrollando y mejorando son las arquitecturas multinúcleo y la programación paralela.

A pesar de la existencia de estos avances, es necesario analizar los problemas en detalle para determinar cuáles son paralelizables y cuáles no, ya que pueden tener tareas que son dependientes del resultado de las anteriores y que por tanto no se pueden llevar a cabo de forma simultánea. Una vez realizado este análisis, si se determina que el programa es potencialmente paralelizable es necesario determinar qué tipo de paralelismo es mejor, así como que modelo de programación paralela es el más adecuado para aplicar en ese caso.

En este capítulo se expone la motivación que ha llevado a la realización de este proyecto (Sección 1.1, *Motivación del Trabajo*), los objetivos que se quieren cumplir con él (Sección 1.2, *Objetivos*) y por último la estructura que sigue el resto del documento (Sección 1.3, *Estructura del documento*).

1.1. Motivación del Trabajo

La creciente demanda de mejora de la eficiencia y velocidad de procesado causada por el aumento en la cantidad de datos que se deben analizar, así como la necesidad ingente de ejecutar programas cada vez

más grandes y complejos ha llevado a la aparición de la computación de alto rendimiento (HPC). Esta normalmente se relaciona con máquinas multinúcleo capaces de funcionar por encima de un TFLOP como los supercomputadores y grupos de ordenadores (*clusters*).

Los supercomputadores y los *clusters* son una buena forma de aumentar el rendimiento y la eficiencia de la ejecución, pero al ser ordenadores tan potentes también conllevan un gasto de energía muy alto. Los supercomputadores actuales llegan a alcanzar 1 PFLOP, esto supone un gasto de energía en el rango de 5 a 10 millones de dólares anuales (entre 4 y 8 millones de euros) [3]. Tal es la cantidad de energía, que muchas de las plantas eléctricas actuales no son capaces de generar esa suma, lo que ha llevado a la búsqueda de alternativas y una de ellas es la programación o computación paralela.

La computación paralela es un modelo de programación que consiste en la ejecución simultánea de diferentes secciones de los programas con el objetivo de reducir el tiempo de ejecución total de este para así poder mejorar la eficiencia. Este tipo de computación se basa en la idea de que todo problema se puede dividir en problemas más pequeños, y que esos subproblemas siempre que no sean dependientes unos de otros, se pueden ejecutar de forma paralela o concurrente.

Se pueden distinguir tres tipos de paralelismo:

- El *Paralelismo de datos* consiste en realizar una misma tarea sobre distintos conjuntos de datos. Se divide el conjunto de los datos en porciones y se asigna cada una de ellas a un núcleo distinto para que éste pueda aplicar una misma instrucción sobre cada porción de datos. Es muy importante que los datos no dependan unos de otros, ya que las dependencias pueden producir errores.
- El *Paralelismo de tareas* consiste en asignar a cada uno de los procesadores o núcleos de los que se dispone una tarea diferente, de forma que los núcleos puedan aplicar las tareas de forma simultánea sobre el conjunto de datos. Antes de aplicar este tipo de paralelismo es necesario comprobar que no haya ningún tipo de dependencia entre las tareas que se quieren aplicar.
- El *Paralelismo de streaming* es diferente a los anteriores ya que se da cuando la entrada o los datos sobre los que se quiere aplicar la operación no tienen una longitud fija debido a que no están disponibles en su totalidad desde el principio, sino que van llegando de forma progresiva para ser procesados.

Como ya se ha comentado, es necesario realizar un análisis de los programas que se quieren paralelizar para evitar caer en errores provocados por dependencias, pero estos no son los únicos errores que pueden darse cuando hablamos de computación paralela. Otros problemas con los que nos podemos encontrar son condiciones de carrera, interbloqueos (*deadlocks*), etc. Esta clase de problemas no son fáciles de solucionar y muchas veces son debidos a la comunicación y sincronización de los hilos implicados, por lo que es necesario el uso de algoritmos de exclusión mutua tales como *mutex* y semáforos.

Los patrones de paralelismo son una forma de evitar que el usuario tenga que lidiar con esos problemas, ya que son los propios patrones los que se ocupan de la creación, sincronización y manejo de los hilos que se van a usar. Estos se podrían considerar como una caja negra, ya que el usuario no sabe realmente lo que están haciendo internamente.

Existen numerosos lenguajes de programación concurrente, APIs y modelos de programación paralela creados expresamente para la programación sobre arquitecturas paralelas. En este caso el trabajo de fin de grado se centra en los modelos de programación paralela y en concreto los que se usan para la evaluación son: *Pthreads, ISO C++ Threads, OpenMP, TBB, FastFlow y GrPPI* (se explican en la Sección 2.2).

1.2. Objetivos

Ante la creciente necesidad de paralelizar los programas para obtener mejores resultados en términos de rendimiento, el uso de patrones de diseño paralelos aparece como una solución para expresar el paralelismo de las aplicaciones de una manera simple que permita un mejor mantenimiento de las mismas.

En este contexto, el objetivo del presente trabajo es comprobar el impacto que tiene sobre el rendimiento el uso de patrones de diseño paralelos para expresar las aplicaciones. Para alcanzar este objetivo se evaluará el rendimiento de un conjunto de aplicaciones de un *benchmark* bien conocido comparando el rendimiento del uso de patrones frente a otras alternativas.

Este objetivo se materializa en los siguientes objetivos concretos:

- Escoger un *benchmark* de evaluación de aplicaciones paralelas que esté ampliamente aceptado y que permita comparar los resultados de los distintos modelos de programación paralela. Dentro de este *benchmark* se seleccionarán las aplicaciones más relevantes para la evaluación, teniendo en cuenta que deben ser de distintos dominios de actuación para conseguir un análisis más representativo.

- Desarrollar versiones de las aplicaciones seleccionadas que expresen el paralelismo en términos de patrones de diseño paralelos utilizando el *framework GrPPI*.

- Realizar una evaluación del rendimiento de las aplicaciones seleccionadas que permita comparar el impacto de los distintos modelos de programación frente al uso de patrones de diseño paralelos.

1.3. Estructura del documento

- Capítulo 1, *Introducción*, contiene los objetivos y la motivación de este trabajo de fin de grado.
- Capítulo 2, *Estado del Arte*, expone los conocimientos previos que se deben tener para entender este proyecto: conocimientos sobre las arquitecturas paralelas existentes, algunos modelos de programación paralela y los *benchmarks* que se podrían usar para evaluar el rendimiento de cada uno de los modelos de programación paralela propuestos.
- Capítulo 3, *Descripción del Problema*, establece los requisitos del proyecto y la matriz de trazabilidad que expone cuales de dichos requisitos se han conseguido.
- Capítulo 4, *Diseño e Implementación*, contiene una descripción detallada de las aplicaciones del *benchmark* de P^3ARSEC que se han seleccionado para la evaluación (en qué consisten, la implementación secuencial y la implementación paralela con cada uno de los modelos de programación seleccionados).
- Capítulo 5, *Evaluación del Rendimiento*, describe la arquitectura que se ha usado a la hora de realizar la evaluación y las metodologías seguidas para sacar los tiempos de ejecución. Además, contiene el análisis de los tiempos obtenidos para las aplicaciones seleccionadas.
- Capítulo 6, *Marco Regulador*, trata los estándares técnicos y restricciones legales que se pueden aplicar.
- Capítulo 7, *Planificación*, plantea las actividades que se han tenido que realizar durante el desarrollo del proyecto, así como la planificación de todas ellas en el tiempo.
- Capítulo 8, *Entorno Socio-Económico*, contiene el presupuesto estimado para este proyecto y el impacto económico que podría tener.
- Capítulo 9, *Conclusiones*, explica cuáles son los objetivos que se han cumplido de los que se proponen en la Sección 1.2 y describe brevemente cuales serían las líneas futuras de trabajo que se podrían seguir.
- Capítulo 10, *English Summary*, resume el trabajo de fin de grado en inglés.

Capítulo 2

Estado del Arte

En este segundo capítulo se presenta el estado del arte, en el que se exponen los conocimientos previos que se deberían conocer antes de empezar a explicar la parte central del trabajo. En la Sección 2.1 se definen brevemente las arquitecturas paralelas que se utilizan actualmente, así como el tipo de arquitectura que se ha usado en la evaluación. La Sección 2.2 describe brevemente algunos de los modelos de programación paralela que existen actualmente, los cuales se han clasificado en dos subsecciones de acuerdo con el tipo de modelo al que pertenecen (básicos (2.2.1) o basados en patrones paralelos (2.2.2)). La Sección 2.3 expone algunos de los *benchmarks* de evaluación de modelos de programación paralela que existen y que por tanto podrían haberse usado en la evaluación, así como el que se ha seleccionado entre todos ellos. Finalmente, se presenta un resumen de los aspectos principales del capítulo (Sección 2.4, *Resumen*).

2.1. Arquitecturas Paralelas

Desde la aparición del primer microprocesador (Intel 4004) en 1971, los ordenadores han ido evolucionando con el objetivo de mejorar su rendimiento. El primer método para mejorar los procesadores fue mediante la reducción del tamaño de los transistores, de forma que en el mismo espacio se podían introducir un mayor número de estos. Este cambio supuso un aumento en la velocidad y eficiencia de los ordenadores, pero ese aumento del número de transistores también supuso un aumento en la cantidad de calor que producía el ordenador y que se debía disipar, lo que provocó un enorme aumento de la energía necesaria para enfriar dicho ordenador. Por esta razón se cambió de estrategia y se pasó a replicar el número de chips creando de esta forma los procesadores multinúcleo o *multi-core*.

Michael J.Flynn estableció uno de los primeros sistemas de clasificación de las arquitecturas de computadores en el artículo [4]. En concreto Flynn afirma que existen cuatro tipos de arquitecturas de acuerdo

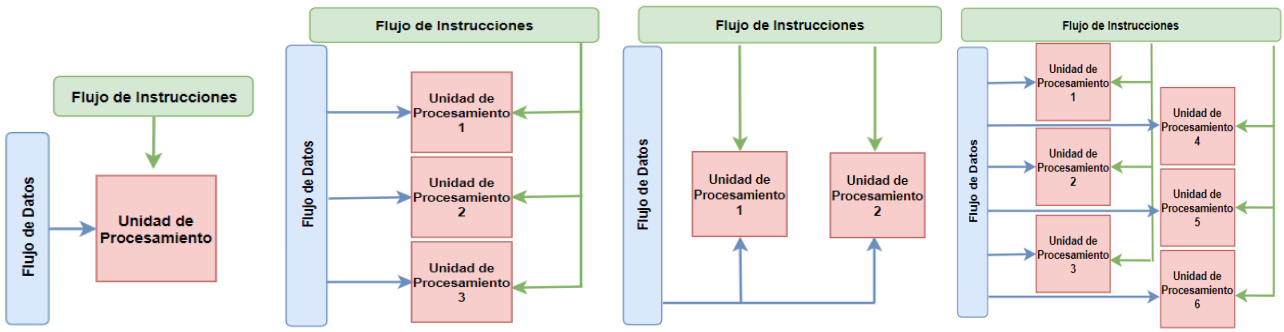
con el número de instrucciones y datos que se ejecutan de forma simultánea.

- *SISD, Single Instruction Single Data:* esta arquitectura no explota ni el paralelismo de datos ni el de tareas, debido a que ejecuta una instrucción sobre un dato en cada ciclo de reloj. Esto se puede ver de manera gráfica en la figura 2-1a, donde el cuadrado azul representa el conjunto de datos, el verde las instrucciones y el rojo la unidad de procesamiento. En este caso solo es necesaria una unidad de procesamiento porque solo se lleva a cabo una ejecución en cada instante de tiempo.

- *SIMD, Single Instruction Multiple Data:* esta arquitectura explota el paralelismo de datos ya que todas las unidades de procesado ejecutan una misma instrucción sobre distintos datos. Este segundo caso se puede ver gráficamente en la figura 2-1b, en la que cada uno de los cuadros de colores representan lo mismo que en el caso anterior. Se puede ver como hay varias unidades de procesamiento cada una de ellas con un dato diferente (puesto que cada una de las flechas sale del conjunto de datos de forma independiente) sobre los que se ejecuta una misma instrucción, debido a que todas las flechas se bifurcan desde la misma línea.

- *MISD, Multiple Instruction Single Data:* esta arquitectura no es muy común, aunque si se ha propuesto en algunas arquitecturas teóricas pues produce un paralelismo redundante al ejecutar diversas instrucciones sobre un mismo dato. Este tercer caso se puede observar en la figura 2-1c, como se puede ver hay varias unidades de procesado que ejecutan distintas instrucciones ya que salen distintas flechas del conjunto de instrucciones (cuadro verde) sobre el mismo dato, puesto que es la misma flecha que sale del cuadro azul y se bifurca.

- *MIMD, Multiple Instruction Multiple Data:* este último tipo de arquitectura consiste en distintos procesadores que ejecutan de forma autónoma instrucciones diferentes sobre distintos datos. Esta viene representada por la figura 2-1d en la que se pueden ver diversas unidades de procesado a las cuales llegan flechas distintas tanto desde el conjunto de datos como desde el conjunto de instrucciones.



(a) *SISD* (b) *SIMD* (c) *MISD* (d) *MIMD*

Figura 2-1: *Taxonomía de Flynn*

Los procesadores multinúcleo puede tener más de un socket dando lugar a una arquitectura multi-socket. Estos sockets son capaces de acceder tanto a los distintos cores o núcleos de los que está compuesto el procesador como a los espacios de memoria de cada uno de esos núcleos, dando lugar a una arquitectura NUMA.

El diseño de memoria NUMA permite que el proceso acceda tanto a la memoria local del propio núcleo como a la memoria no local, constituida por la memoria compartida entre todos los procesadores y la memoria local de otro procesador. Sin embargo, el acceso a la memoria local conlleva menos tiempo que el acceso a la memoria no local, de forma que cuando un proceso trata de acceder a la memoria no local el tiempo de ejecución se ve incrementado.

2.2. Modelos de Programación Paralela

Un modelo de programación paralela se puede definir como un conjunto de abstracciones que permiten al programador tener una visión simplificada de la arquitectura, facilitando la referencia de los componentes lógicos del programa con los físicos de la arquitectura, lo que supone una mayor explotación del paralelismo.

Hay numerosos modelos de programación paralela adaptados para ser usado en conjunto con uno o varios lenguajes de programación diferentes. Como el lenguaje de programación seleccionado para la evaluación es C++ se han seleccionado simplemente modelos que estuvieran disponibles para dicho lenguaje de programación.

Existen varias clasificaciones para los modelos de programación, pero en este caso se van a dividir en dos clases: modelos de programación paralela básicos (Sección 2.2.1) y modelos de programación paralela basados en patrones paralelos (Sección 2.2.2).

2.2.1. Modelos de Programación Paralela Básicos

Los modelos de programación paralela básicos son aquellos que se basan en primitivas y que pueden ser tanto de alto nivel como de bajo nivel. Los modelos de programación de bajo nivel son aquellos que están ligados a la plataforma en la que se ejecutan, por ello requieren por parte del programador un mayor conocimiento de las características de ésta. Por otro lado, los modelos de programación de alto nivel proporcionan abstracciones de la plataforma, debido a lo cual no son tan dependientes de ésta y aumentan el nivel de portabilidad, haciendo posible de esta forma que se pueda ejecutar el mismo código en otras plataformas con distintas características.

Para una mejor comprensión de las diferencias entre los distintos modelos de programación paralela se va a utilizar el código secuencial de la subrutina *daxpy* (Listado 2.1) y se va a adaptar para cada uno de

los modelos que se proponen:

```
void daxpy ( int n, double a, double *x, double *y, double *z) {

     int i;
     for (i=0; i<n; i++){ //Inicializa los vectores
       x[i] = (double)i * (double)i;
       y[i] = (i+1.)*(i-1.);
     }
     for (i=0; i<n; i++){
       z[i] += a * x[i] + y[i]; //Computo
     }

}
```

Listado 2.1: *Código secuencial para la subrutina daxpy.*

A continuación, se describen brevemente algunos de los *modelos de programación de bajo nivel* que soportan C++:

ISO C++ Threads

Los hilos de C++ vienen definidos en la clase *thread* y están disponibles a partir de la versión C++11 como se puede ver en el estándar [5]. Estos hilos permiten llevar a cabo diversos procesos de forma simultánea, comenzando su ejecución justo después de la creación del objeto y ejecutando el código de la función asociada a dicho hilo. Dependiendo de la opción de sincronización (*join o detach*) asociada a la finalización del procesado de los hilos, estos pueden seguir dos patrones: si la opción es *join* el hilo debe esperar hasta que el resto de hilos terminen su cómputo, por otro lado, si la opción es *detach* el hilo es independiente del resto y por tanto, en el momento que finalice su tarea terminará.

Para sincronizar los hilos en el caso de que fuera necesario se pueden usar las funciones *std::mutex y std::atomic*, gracias a las cuales se pueden evitar las condiciones de carrera producidas por ejemplo cuando más de un hilo trata de acceder a la misma posición de memoria al mismo tiempo.

En el Listado 2.2 se puede observar el código de la subrutina *daxpy* adaptado para su ejecución usando hilos de C++.

PThreads

Pthreads o *POSIX Threads* es un conjunto de interfaces definidas en la clase *pthread*.

Esta biblioteca es una *API* basada en los estándares de los lenguajes de programación C/C++. La

```
auto daxpy = [&](int start, int end) -> void{
 for (int i = start; i < end; i++)
    y[i] = a * x[i] + y[i];
};

std::thread threads[n_threads];
int chunk_per_thread = N / n_threads;
int rest = N \% n_threads;
for(int j=0; j<n_threads; j++){

    int init = chunk_per_thread*j;
    int end;
    if (j==nthreads-1){
        end = init + chunk_per_thread + rest;
    }
    else {
        end = init + chunk_per_thread;
    }
    threads[j] = std::thread(daxpy, init, end);
}
```

Listado 2.2: *Código para la subrutina daxpy usando ISO C++ Threads.*

biblioteca *Pthreads* crea y destruye los hilos mediante el uso de las siguientes funciones: *pthreads_create, pthread_join y pthread_exit.*

Para poder sincronizar y comunicar los hilos de forma que no se produzcan errores, este modelo al igual que el anterior, necesita el uso de *mutexes* y semáforos.

En el Listado 2.3 se puede ver el código de la subrutina *daxpy* para su ejecución en paralelo usando la biblioteca *Pthreads.*

CUDA

CUDA (*Compute Unified Device Architecture*) hace referencia a una arquitectura de procesamiento paralelo compuesta por un conjunto de herramientas y un compilador. Este modelo fue desarrollado por nVidia en 2007 y permite desarrollar código en C/C++ para ser ejecutado sobre las tarjetas gráficas (*GPUs*) de nVidia [6].

El modelo de programación de CUDA utiliza de forma conjunta tanto la CPU como la GPU, por lo que el código de CUDA está compuesto por dos partes: una parte de código secuencial que se ejecutará en la CPU (también denominado *host*) y la otra parte de código paralelo para GPU (o *device*). Dentro del código secuencial se llama al *kernel*, que puede ser una función o incluso un programa completo y que se procesa en paralelo en la GPU mediante el uso de hilos. La estructura de las tarjetas gráficas viene definida

```
auto daxpy = [&](int start, int end) -> void{
 for (int i = start; i < end; i++)
    y[i] = a * x[i] + y[i];
};

pthread_t threads[n_threads];
int chunk_per_thread = N / n_threads;
int rest = N \% n_threads;
for(int j=0; j<n_threads; j++){

    int init = chunk_per_thread*j;
    int end;
    if (j==nthreads-1){
        end = init + chunk_per_thread + rest;
    }
    else {
        end = init + chunk_per_thread;
    }
    pthread_create(&threads[j], NULL, daxpy, &init, &end);
}
```

Listado 2.3: *Código para la subrutina daxpy usando PThreads.*

por una serie de mallas (*grids*) compuestas por bloques de hilos independientes (con un máximo de 512 hilos por bloque), que se pueden ejecutar en paralelo.

En el Listado 2.4 se muestra el código correspondiente a la subrutina *daxpy* usando CUDA.

```
__global__ void daxpy(double a, double *x, double *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) y[i] += a*x[i];
}
...
int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);
daxpy<<<n_bloques, n_hilos_por_bloque>>>(N, 2.0, x, y);
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

Listado 2.4: *Código de la subrutina daxpy usando CUDA.*

OpenCL

OpenCL (*Open Computing Language*) es un estándar abierto para programación en plataformas heterogéneas, este fue creado por el grupo Khronos basándose en una propuesta previa realizada por Apple en colaboración con los líderes de la industria [7]. OpenCL es parecido a CUDA, aunque éste a diferencia del anterior no está simplemente limitado a las tarjetas gráficas de nVidia, sino que se puede usar con una

gran variedad de GPUs como las de Intel, AMD, IBM, Apple, etc.

Este modelo permite la ejecución del código generado solo en la CPU, solo en la GPU, en ambas o incluso en múltiples GPUs. La plataforma OpenCL consta de un *host* (CPU) y uno o varios *devices* (GPUs) de OpenCL, compuestos por varias unidades de computo, que a su vez se dividen en elementos de procesamiento.

OpenCL proporciona una abstracción del hardware para los *devices* que incluye modelos de *hardware*, memoria y programación paralela. Además, cabe destacar que estos *devices* no pueden ejecutar código tradicional como el que se haría con el lenguaje C++, sino que provee a los programadores de un lenguaje de *kernel* específico para los dispositivos OpenCL [8].

En el Listado 2.5 se puede observar el código de la subrutina *daxpy* en el caso de usar el estándar OpenCL.

```
__kernel void DAXPY (__global float* x, __global float* y, float a){
    const int i = get_global_id (0);
    y [i] += a * x [i];
}
```

Listado 2.5: *Código de la subrutina daxpy usando OpenCL.*

Una vez descritos los modelos de bajo nivel seleccionados, se van a exponer algunos *modelos de programación de alto nivel*, que al igual que los anteriores también soportan el lenguaje de programación C++:

OpenMP

OpenMP (Open Multi-Processing) es una interfaz de programación de aplicaciones que permite la programación multiproceso con memoria compartida en múltiples plataformas. Este se basa en una serie de directivas de compilación o *pragmas*, las cuales permiten aplicar paralelismo sobre bloques de código iterativo [9]. Al ser un modelo basado en memoria compartida las variables que se usan durante el proceso se comparten entre los distintos hilos. Este hecho puede llevar a condiciones de carrera, por ello OpenMP implementa unas primitivas especiales que permiten especificar qué variables son compartidas y cuáles privadas.

Un aspecto clave a tener en cuenta al tratar con la biblioteca OpenMP son las distintas políticas de planificación de tareas de las que consta (*static, dynamic y guided*). La política que se usa por defecto es la estática que consiste en la asignación de los distintos *chunks* (trozos) a los hilos mediante *round-robin*,

es decir, si se sigue esta política cada hilo ejecutará la misma cantidad de *chunks*.

En la política dinámica los *chunks* se van entregando a los distintos hilos a medida que terminan las tareas asignadas, es decir, que conlleva una asignación mediante demanda. Y, por último, en la guiada se comienza con un tamaño de bloque grande y se va disminuyendo, en esta política cada hilo va cogiendo un bloque hasta que todos ellos han sido procesados.

En el Listado 2.6 se puede ver el código que se usaría en el caso de querer paralelizar la subrutina *daxpy* usando OpenMP.

```
void daxpy (int n, double a, double *x, double *y){
    #pragma omp parallel for default(none) shared(n,x,y,a,z)
    for (int i=0; i<n; i++){
        y[i] = a * x[i] + y[i];
    }
}
```

Listado 2.6: *Código de la subrutina daxpy usando OpenMP.*

TBB

TBB (*Threading Building Blocks*) es una biblioteca diseñada por Intel, que se usa en gran medida junto con el lenguaje de programación C++. Esta biblioteca se utiliza para la programación paralela con memoria compartida y para la computación heterogénea [10].

Este modelo consta de una serie de algoritmos y estructuras de datos gracias a las cuales se pueden paralelizar los programas, lo que permite que el programador sea capaz de olvidarse de los problemas derivados de la creación, sincronización y destrucción de los hilos en ejecución, actividades que suelen ser dependientes del sistema. Esta característica es fundamental ya que lleva a que se considere TBB como una biblioteca capaz de separar la programación de las características de la máquina en la que se ejecuta.

En TBB se implementa una función denominada *task stealing* (robo de tareas) que se utiliza con el fin de balancear la carga de trabajo que recae sobre cada uno de los núcleos incrementando de esta forma el aprovechamiento y la escalabilidad.

En el Listado 2.7 se muestra el código paralelo de la subrutina *daxpy* usando la biblioteca TBB.

Parallel STL

Parallel STL (*Parallel Standard Template Library*) es una implementación paralela de los algoritmos de la biblioteca estándar de C++, a los cuales se les añade un nuevo argumento que corresponde con

```
void daxpy_tbb(double a, std::vector<double> &x, std::vector<double> &y) {

    tbb::parallel_for(tbb::blocked_range<int>(0,x.size()),
    [&](tbb::blocked_range<int> r){
        for (size_t i=r.begin();i!=r.end();++i) {
            y[i]=a *x[i]+y[i];
        }
    });
}
```

Listado 2.7: *Código de la subrutina daxpy usando TBB.*

las políticas de ejecución [11]. La biblioteca STL consta de más de 100 algoritmos de búsqueda, conteo y manipulación de rangos y los elementos que los componen. Estos algoritmos se pueden invocar mediante una llamada a la política de ejecución, que puede ser secuencial, paralela, paralela vectorizada, vectorizada secuencial o simplemente vectorizada [12].

En el Listado 2.8 se puede observar el código correspondiente a la versión paralela usando *Parallel STL* de la subrutina *daxpy*.

```
void daxpy(size_t n, float a, const float* x, const float* y){
 using namespace agency;
 bulk_invoke(par(n), [=](parallel_agent &self) {
     int i = self.index();
     y[i] = a * x[i] + y[i];
 });
}
```

Listado 2.8: *Código para la subrutina daxpy usando Parallel STL.*

2.2.2. Modelos de Programación Paralela basados en Patrones Paralelos

Por otro lado, los modelos de programación basados en patrones paralelos en lugar de usar primitivas como los anteriores, hacen uso de estructuras de orden superior que se encargan de manera automática de comunicar y sincronizar los procesos facilitando de esta forma el trabajo del programador. Hay muchos patrones de paralelismo como se puede ver en el artículo [13], pero solo se van a explicar los más usados. Estos patrones se pueden dividir en tres grupos principales, de acuerdo con el tipo de paralelismo que explotan de los explicados en la Sección 1.1:

2.2.3. Patrones Paralelos de Datos

Los patrones paralelos de datos son aquellos que como su propio nombre indica explotan el paralelismo de datos. Dentro de este tipo de patrones se pueden encontrar: *Map, Reduce, Fork, Map-Reduce y Stencil.*

Map es un patrón que permite aplicar una misma operación o instrucción sobre cada uno de los elementos de una lista de forma simultánea, generando una nueva lista con los resultados obtenidos de aplicar la operación, que será la salida del patrón. El Map como ya se ha comentado explota el paralelismo de datos, pero además concuerda con la arquitectura SIMD explicada en la Sección 2.1.

Parallel_for es un patrón muy similar al anterior, aunque en este caso en lugar de tener un vector como entrada se establece un rango de actuación ya que se le pasan como parámetros el valor inicial y final de dicho rango, de esta forma el patrón itera por los distintos valores intermedios ejecutando las operaciones en su interior.

Reduce es un patrón que aplica una operación sobre cada par de elementos de una lista y se guarda cada uno de los resultados parciales con el objetivo de combinar todos los valores de dicha lista, teniendo como resultado un único valor. Esa operación puede ser cualquiera, aunque la más común es la suma.

Fork es un patrón similar al Map, pero en este caso en lugar de aplicar la misma operación sobre cada uno de los elementos de la lista de entrada se aplican operaciones diferentes. Este patrón de paralelismo concuerda con la arquitectura MIMD explicada en la Sección 2.1.

Map-Reduce es un patrón que surge de la combinación de los patrones anteriores Map y Reduce. Este consiste en el uso del patrón Map sobre uno o varios conjuntos de datos y luego la aplicación del patrón Reduce sobre los elementos de la lista resultante del patrón anterior.

Stencil es un patrón que permite aplicar una operación sobre cada uno de los elementos de una lista, teniendo como datos para dicha operación el propio elemento y un vecindario del conjunto de entrada.

2.2.4. Patrones Paralelos de Tareas

Los patrones paralelos de tareas a diferencia de los anteriores explotan el paralelismo de tareas. Dentro de este tipo de patrones se pueden encontrar:

Divide & **Conquer** es un patrón que permite dividir un problema complejo en subproblemas más sencillos una y otra vez hasta que se llegue a un conjunto de problemas que se puedan resolver de forma sencilla. Una vez resueltos se combinan los resultados obtenidos para todos ellos y se obtiene la solución final para el problema inicial.

Branch & **Bound** es un patrón que divide de forma recursiva el espacio de búsqueda y extrae los elementos resultantes de los subespacios mediante el uso de una función objetivo. La elección de una buena función objetivo es un paso fundamental ya que puede determinar la efectividad de todo el patrón.

2.2.5. Patrones Paralelos de Streaming

Los patrones paralelos de streaming se llaman de esa forma porque tratan de explotar el paralelismo de streaming. Dentro de este tipo hay patrones como:

Pipeline es un patrón en el que el cómputo se realiza en diferentes etapas (sin número límite de estas). Este siempre consta al menos de dos etapas, una primera etapa de creación en la que se generan los datos mediante una operación de generación y una segunda etapa a la que se le van pasando los datos a medida que se van generando. Esta segunda etapa puede ser una etapa intermedia, que son aquellas que aplican diferentes operaciones sobre los datos obtenidos de la etapa anterior y una vez realizado el cálculo se pasan los resultados a la siguiente etapa, o puede ser una etapa de consumición, que son las etapas finales del Pipeline, éstas al igual que las intermedias aplicarían una operación, pero no devolverían ningún resultado ya que no puede haber ninguna otra etapa después.

Farm es un patrón que permite la aplicación de forma paralela (mediante el uso de hilos) de operaciones sobre un conjunto de datos. El Farm tiene la habilidad de asignar a los diferentes nodos de cómputo cada una de las tareas, lo que permite la explotación del paralelismo de tareas. Este patrón suele ser componible, es decir, que se puede usar en conjunto con otros patrones como por ejemplo el Pipeline.

Filter es un patrón de filtrado que se queda solo con los elementos de entrada que cumplen su predicado. Este patrón se podría considerar como una sentencia *if* que solo deja pasar aquellos datos que cumplen la condición para que se les pueda aplicar una operación.

Iteration es un patrón que permite la aplicación de bucles de operaciones sobre cada uno de los elementos de entrada hasta que se cumple un predicado. Este patrón se podría considerar equivalente a las sentencias *while*, ya que esa sentencia ejecuta una misma operación siempre que se siga cumpliendo la condición, aunque en el caso del patrón Iteration este sigue ejecutándose hasta que se cumpla la condición.

Reduction es el patrón equivalente al Reduce de la Sección 2.2.3 aunque en este caso en lugar de aplicarse sobre un conjunto finito se aplica sobre un conjunto de datos que se va incrementando de forma progresiva mientras se va ejecutando el patrón.

2.2.6. Modelos

Hay una gran cantidad de modelos de programación basados en patrones paralelos como se puede ver en el artículo [13], aunque solo se van a explicar aquellos que soportan el lenguaje de programación C++, como ya se ha comentado.

ASSIST

ASSIST (A Software development System based upon Integrated Skeleton Technology) es un modelo de programación con un enfoque unificado, destinado a la implementación de aplicaciones paralelas y distribuidas de alto rendimiento. Este modelo permite al programador diseñar aplicaciones como si fueran grafos genéricos en los que los nodos representan los componentes paralelos y los arcos que unen dichos nodos las interfaces abstractas. Cada uno de esos componentes o nodos se representan mediante un módulo ASSIST que puede ser de dos tipos: secuencial o paralelos (*parmod*) [14].

En el Listado 2.9 se muestra el código paralelo de la subrutina *daxpy* usando el modelo de programación basado en patrones de diseño paralelos ASSIST.

```
parmod daxpy(input_stream double x[N], double y[N], int a){
    proc dxp(in double x[N], double y[N], int a)
    $c++{ register double r=0.0;
        for(register int i=0; i<N; i++)
            r = a * x[i] + y[i];
        assist_out(y,r);
    }c++$
}
```

Listado 2.9: *Código para la subrutina daxpy usando ASSIST.*

MUESLI

MUESLI (MUEnster Skeleton LIbrary) es una biblioteca de patrones desarrollada por la Universidad de Muenster, que explota tanto el paralelismo de tareas como el de datos. MUESLI se basa en las plantillas y características del lenguaje de programación de C++, con el objetivo de hacer patrones más sencillos y fáciles de usar por parte del programador.

Este modelo de programación se basa principalmente en cuatro patrones de paralelismo que son: *Pipeline, Farm, Divide & Conquer y Branch & Bound*, cuya funcionalidad es la misma que la que ya se ha descrito anteriormente [15].

En el Listado 2.10 se puede observar cómo sería la implementación de la subrutina *daxpy* si se usara la

biblioteca de patrones MUESLI para su paralelización.

```
const DistributedArray<double> daxpy(const DistributedArray<double>&x,
const DistributedArray<double>&y, int a){
    for (int k=0; k<x.getLocalSize(); k++){
        y.set(k,(a * x.get(k) + y.get(k);
    }
    return y;
}
```

Listado 2.10: *Código de la subrutina daxpy usando el modelo MUESLI.*

SkePu

SkePU es una biblioteca implementada mediante plantillas de C++ que proporciona patrones para GPU gracias al uso de CUDA y OpenCL, lo que permite realizar cómputo que explota el paralelismo de datos. Este modelo de programación consta de una implementación secuencial para su ejecución en CPU y otra paralela con OpenMP [16].

SkePU consta de varios patrones de paralelismo: *Map, Reduce, Map-Reduce, Map-Overlap y Map-Array*, siendo la funcionalidad de los tres primeros la que ya se ha expuesto cuando se explicaban los tipos de patrones paralelos y los dos últimos serían variaciones del patrón *Map*.

En el Listado 2.11 se puede ver el código paralelo para la subrutina *daxpy* usando la biblioteca SkePU.

```
float daxpy_unary(int a, double x, double y){
       return a * x + y;
}
Vector<double> daxpy(int a, Vector<double> &v1, Vector<double> &v2){
       auto dxp = Map<3>(daxpy_unary);
       Vector<double> result(v1.size());
       return dxp(result, a, v1, v2);
}
```

Listado 2.11: *Código para la subrutina daxpy usando el patrón Map del modelo SkePU.*

SKeTo

SkeTo (Skeletons in Tokyo) es una biblioteca implementada usando el lenguaje de programación C++ en conjunto con MPI destinada a la programación distribuida. El objetivo de los desarrolladores de SkeTo era que los programadores no tuvieran la sensación de estar haciendo código paralelo debido a la simplicidad que aportaba la biblioteca.

Este modelo proporciona patrones paralelos que explotan el paralelismo de datos en las listas, matrices y árboles. Algunos de los patrones de paralelismo que implementa son: *Map, Reduce, Scan, Dist, Gather, Rot, Shift, Zip y Zipwith* [17].

En el Listado 2.12 se muestra el código correspondiente con la subrutina *daxpy* cuando se paraleliza usando la biblioteca SkeTo.

```
auto daxpy = zip(
    [](float x, float y, float a) {
        return a*x+y;
    }
);
```

Listado 2.12: *Código de la subrutina daxpy usando el patrón zip del modelo SKeTo.*

FastFlow

FastFlow es un marco de programación paralela basado en las plantillas del lenguaje de programación C++ e implementado sobre la biblioteca de *Pthreads* [18]. Este modelo está enfocado a la explotación tanto del paralelismo de datos como el de streaming. FastFlow se creó para su uso en plataformas heterogéneas paralelas, en concreto para aquellas formadas por *clusters* de memoria compartida.

Fastflow ha sido diseñado con tres niveles, el más bajo de ellos implementa una cola libre de cerrojos con un consumidor y un productor. El segundo nivel implementa colas tanto de un productor y múltiples consumidores, como de múltiples productores y un único consumidor. Y, por último, el tercer nivel es el que provee a los programadores de los patrones paralelos [19].

Este modelo tiene como elemento principal para sus ejecuciones los *ff_nodes*, estos nodos contienen código secuencial en su interior y se pueden ejecutar de forma paralela, es decir, aunque el código de su interior no se pueda paralelizar, el código de distintos nodos se puede ejecutar de forma simultánea. Cada instancia de *ff_node* consta de tres métodos: *svc* el cual contiene el computo que se quiere llevar a cabo, *svc_init y svc_end*.

Los principales patrones de paralelismo de este modelo se pueden dividir en patrones de streaming como *Pipeline (ff_Pipe) y Farm (ff_Farm)* y patrones de datos como *ParallelFor (parallel_for), ParallelForReduce (parallel_reduce) y ParallelForPipeReduce (parallel_reduce_idx)* [20].

En el Listado 2.13 se puede observar el código de la subrutina *daxpy* paralelizado mediante el uso del *framework* FastFlow.

```
pf = new ff::ParallelFor(n_threads, false, true);
pf->parallel_for(0, N, 1, 1,
    [&](const int index) {
        return a * x[index] + y[index];
    }
    , n_threads
);
```

Listado 2.13: *Código para la subrutina daxpy usando el patrón parallel_for del modelo FastFlow.*

GrPPI

GrPPI (Generic reusable Parallel Pattern Interface) es una biblioteca implementada mediante plantillas de C++ que permite evaluar de forma sencilla la eficiencia de distintos modelos de programación paralela sin necesidad de cambiar líneas del código. Este modelo no aporta simplemente unos patrones implementados de forma paralela, sino que esos patrones tienen distintas implementaciones internas usando distintos modelos de programación básicos tanto de alto como de bajo nivel. De esta forma el programador podrá determinar qué modelo de programación es el más adecuado en cada caso, sin modificar apenas la implementación [21].

En el Listado 2.14 se muestra la implementación de la subrutina *daxpy* en el caso de usar la biblioteca de patrones paralelos GrPPI.

```
grppi::map(e, make_tuple(begin(x),begin(y)), end(x), begin(y),
    [a](int vx, int vy) { return a * vx + vy; });
```

Listado 2.14: *Código de la subrutina daxpy usando GrPPI.*

2.3. Benchmarks

Los *benchmarks* están compuestos por una aplicación o conjunto de aplicaciones destinadas a medir el rendimiento de un sistema o alguno de los componentes de éste. Se pueden encontrar diversos *benchmarks* preparados expresamente para la evaluación de paralelismo, es decir, que contienen aplicaciones paralelizables óptimas para medir el rendimiento de distintos modelos de programación paralela. Algunos de esos *benchmarks* son los siguientes:

SPEC

El grupo SPEC (*Standard Performance Evaluation Corporation*) consta de una gran cantidad de *benchmarks* para la evaluación de distintos entornos tales como: *cloud*, CPU, tarjetas gráficas, HPC, aplicaciones del tipo cliente/servidor para Java, servidores mail, almacenaje, potencia, virtualización y servidores web.

A pesar de la gran cantidad de *benchmarks* que ofrece los únicos interesantes en este caso son aquellos que pertenecen al dominio de HPC.

En el área del HPC cuenta con *benchmarks* para evaluar modelos de programación paralela tales como *OpenMP, MPI, OpenCL y OpenACC* como se puede ver en su página web [22].

NPB

El *benchmark* NPB (*NAS Parallel Benchmark*) fue desarrollado por la NASA con el objetivo de evaluar el rendimiento de los supercomputadores mediante el uso de los modelos de programación *OpenMP y MPI*.

Este benchmark consta de cinco *kernels*, tres pseudoaplicaciones, tres aplicaciones multizona (desarrolladas para evaluar el rendimiento de los paradigmas y herramientas de programación multinivel o híbrida), cuatro *benchmarks* de programación no estructurada, entrada/salida paralela y movimiento de datos y cuatro aplicaciones *GridNPB* destinadas a la evaluación de redes computacionales. Para obtener más información de las aplicaciones que lo componen se puede consultar su página web [23].

STAP

El benchmark STAP (Space-Time Adaptive Processing) fue diseñado como una técnica de procesado adaptativa utilizada para cancelar las interferencias de los radares aéreos. Más tarde en 1997 se desarrolló una nueva versión denominada RT-STAP (Real Time Space-Time Adaptive Processing) destinada a la evaluación de aplicaciones de computadores escalables de alto rendimiento en tiempo real para su implementación en plataforma integradas [24].

SPLASH

El *benchmark* SPLASH (*Stanford ParalleL Applications for Shared-Memory*) es un conjunto de aplicaciones desarrollado por la Universidad de Stanford en 1991 con el objetivo de evaluar sistema multiproceso con memoria compartida. Este *benchmark* se ha ido modificando dando lugar a otros más avanzados: *SPLASH-2 y SPLASH-3*. Ambas mejoras del *benchmark* contienen las mismas aplicaciones: *Barnes, Cholesky, FMM, Radiosity, Raytrace y Volrend*, cuyas descripciones se pueden consultar en el artículo que compara ambas versiones [25].

Rodinia

El *benchmark* Rodinia fue desarrollado por la Universidad de Virginia para la computación heterogénea. Éste consta de aplicaciones que dan soporte tanto a CPU como a GPU, ideal para la evaluación de modelos

de programación como *CUDA u OpenCL*.

Este *benchmark* consta de nueve aplicaciones que son: *K-means, Needleman-Wunsch, HotSpot, Back propagation, SRAD, Leukocyte tracking, Breadth-First search, Stream cluster y Similarity scores*. La descripción de dichas aplicaciones se puede encontrar en el artículo publicado por el departamento de informática de la Universidad de Virginia [26]

PARSEC

El *benchmark* de PARSEC está compuesto por una serie de aplicaciones cuidadosamente seleccionadas para conseguir un conjunto lo suficientemente grande y representativo. Sin una selección meticulosa de las aplicaciones para medir el rendimiento se podría llegar a una situación en la que los resultados obtenidos no fueran correctos y por tanto las conclusiones extraídas de dicho experimento no serían válidas.

Este *benchmark* hay ido variando con el tiempo, es decir, que se han ido añadiendo más aplicaciones. En concreto la versión más actual consta de 9 aplicaciones (*blackscholes, bodytrack, facesim, ferret, fluidanimate, freqmine, raytrace, swaptions, vips y x264*) y 3 kernels (*canneal, streamcluster y dedup*) seleccionados entre un amplio rango de dominios, todos ellos paralelizables [27, 28].

P^3ARSEC

Este último *benchmark* es el mismo que el anterior pero con una pequeña variación, que es que algunas de las aplicaciones que provee el *benchmark* tienen además de las versiones paralelas anteriores la versión de *Fastflow*, en concreto esas aplicaciones son: *blackscholes, bodytrack, canneal, dedup, facesim, ferret, fluidanimate, freqmine, raytrace, swaptions y vips*, es decir, todas las aplicaciones y *kernels* menos la aplicación *x264*, ya que como se explica en el artículo [29] era demasiado complejo y no se podían separar de forma sencilla las partes paralelizables.

2.4. Resumen

Como se ha comentado hay numerosos modelos de programación paralela que se podrían emplear para paralelizar las aplicaciones y aunque lo ideal sería evaluar cada uno ellos para determinar cuál es el que funciona mejor para los distintos casos que se han analizado, debido a la extensión que supondría evaluarlos todos solo se han seleccionado cinco modelos de los expuestos en las Secciones 2.2.1 y 2.2.2, que son: *ISO C++ Threads, TBB, OpenMP, FastFlow y GrPPI*. Analizando de esta forma modelos tanto de bajo nivel, como de alto nivel y basados en patrones paralelos, con lo cual se podría determinar qué tipo de modelo es más eficiente para un caso concreto.

De entre la gran variedad de *benchmarks* que se podrían usar para la evaluación se ha escogido el *benchmark* P^3ARSEC que contiene las implementaciones de todos los modelos de programación mencionados, excepto la de *GrPPI* que es la que se ha tenido que implementar. Por esta razón el modelo *GrPPI* se explica con más detalle en la Sección 4.1. Además, también se pueden conocer más detalles del *benchmark* seleccionado y cada una de las aplicaciones elegidas en la Sección 4.2.

Capítulo 3

Descripción del Problema

En este capítulo se exponen los requisitos que deben cumplir las aplicaciones del *benchmark* P^3ARSEC una vez han sido paralelizadas usando el modelo de programación basado en patrones paralelos GrPPI. En la Sección 3.1 se describe el formato que siguen las definiciones de cada uno de los requisitos expuestas en las subsecciones 3.1.1 y 3.1.2. En la Sección *Análisis de los Requisitos* (3.2) se analizan los requisitos de la sección anterior para determinar cuáles se han cumplido y cuales no, mediante el uso de una matriz de trazabilidad.

3.1. Requisitos

Los requisitos generalmente se definen como un conjunto de condiciones o capacidades que debe cumplir el sistema para satisfacer un estándar o una especificación acordada entre el desarrollador y la persona contratante. Los requisitos deben cumplir una serie de características tal como se especifica en el estándar de IEEE [30]: estos deben ser correctos, claros (no ambiguos), completos, consistentes, verificables, modificables y trazables. En la siguiente tabla se muestra la estructura o el formato que van a tener los requisitos:

<table>
<tr><td>ID</td><td>X_Y</td><td>Requisito</td><td>Nombre del Requisito</td></tr>
<tr><td>Necesidad</td><td>Necesario / No Necesario</td><td colspan="2">Texto del Requisito</td></tr>
<tr><td>Prioridad</td><td>Alta / Media / Baja</td><td colspan="2" rowspan="5">Descripción del Requisito</td></tr>
<tr><td>Estabilidad</td><td>Estable / No Estable</td></tr>
<tr><td>Tipo</td><td>Funcional / No Funcional</td></tr>
<tr><td>Estado</td><td>Propuesto / Verificado / Validado / Rechazado</td></tr>
<tr><td>Verificabilidad</td><td>Verificable / No Verificable</td></tr>
</table>

Tabla 3.1: *Ejemplo del formato para las tablas de requisitos*

Como se puede ver la tabla anterior consta de una serie de campos con distintas funcionalidades, los cuales se explican a continuación:

- **ID** → Éste constituye el código de identificación del requisito, este código debe ser inequívoco ya que es el que se usa después para hacer referencia a cada uno de ellos en la matriz de trazabilidad. El ID va a seguir el formato *X_Y*, en el que la *X* identifica el tipo de requisito que es: *FR* en el caso de los requisitos funcionales y *NFR* en el caso de los no funcionales. Por otro lado, la *Y* es el número del requisito que identifica a cada uno de ellos dentro del subconjunto de los funcionales o no funcionales, este empezará con el valor 01 y se irá incrementando a medida que se vayan añadiendo más requisitos.

- **Requisito** → Este campo expone con pocas palabras en que consiste el requisito.

- **Necesidad** → Éste constituye el grado de importancia que tiene el requisito. Hay dos valores posibles: *Necesario*, en el caso de que el requisito sea de obligado cumplimiento para el correcto funcionamiento de las aplicaciones o *No Necesario*, en el caso de que si el requisito no se puede cumplir el funcionamiento de las aplicaciones no se verá afectado.

- **Prioridad** → Éste constituye el valor de la prioridad asociada al requisito. Hay tres posibles valores: *Alta*, *Media* o *Baja*.

- **Estabilidad** → Éste constituye el nivel de estabilidad del requisito. Hay dos valores posibles: *Estable*, si el requisito es fijo y no se puede cambiar o *No Estable*, en el caso de que el requisito se pueda modificar durante el desarrollo del proyecto.

- **Tipo** → Este campo se refiere al tipo de requisito que es: *Funcional* o *No Funcional*.

- **Texto del Requisito** → Éste constituye una descripción más detallada sobre el requisito.

- **Estado** → Este campo hace referencia al estado del ciclo de vida de los requisitos en el que se encuentra. hay cuatro posibles valores: *Propuesto*, *Verificado*, *Validado* y *Rechazado*. Teniendo en cuenta que si un requisito ha sido validado implica que también ha sido verificado.

- **Verificabilidad** → Este campo hace referencia al nivel de verificabilidad que tiene un requisito, es decir si es verificable (*Verificable*) o no (*No Verificable*).

Los requisitos de sistema se pueden clasificar en dos tipos: requisitos funcionales (Sección 3.1.1) y requisitos no funcionales (Sección 3.1.2).

3.1.1. Requisitos Funcionales

Los requisitos funcionales son aquellos que describen las funcionalidades que deben cumplir en este caso las aplicaciones. Estos pueden estar relacionados con las entradas, el comportamiento y las salidas de cada una de las aplicaciones. La forma principal de comprobar que se cumplen este tipo de requisitos es mediante casos de uso.

A continuación, se muestran los requisitos funcionales que se han establecido en este trabajo de fin de grado:

ID	*FR_01*	**Requisito**	*Resultado Blackscholes*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*Los precios de la cartera de opciones obtenidos mediante la versión implementada con GrPPI deben ser iguales a los resultantes del resto de las implementaciones.*	
Estabilidad	*Estable*		
Tipo	*Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.2: *Requisito Funcional FR_01*

ID	*FR_02*	**Requisito**	*Resultado Bodytrack*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*La imagen de salida de la aplicación y el archivo poses.txt obtenidos con la versión de GrPPI deben ser iguales a los resultantes del resto de versiones para las mismas imágenes de entrada.*	
Estabilidad	*Estable*		
Tipo	*Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.3: *Requisito Funcional FR_02*

ID	*FR_03*	**Requisito**	*Resultado Canneal*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*El coste de enrutamiento resultante de la versión implementada con GrPPI debe estar en el rango* $[9,03004 \times 10^9; 9,03452 \times 10^9]$ *como pasa con el resto de las versiones.*	
Estabilidad	*Estable*		
Tipo	*Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.4: *Requisito Funcional FR_03*

ID	*FR_04*	**Requisito**	*Resultado Facesim*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*Todos los archivos deformable-object generados por la versión de GrPPI deben ser iguales a los obtenidos con el resto de las implementaciones.*	
Estabilidad	*Estable*		
Tipo	*Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.5: *Requisito Funcional FR_04*

ID	*FR_05*	**Requisito**	*Resultado Ferret*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*El ranking resultante para cada una de las imágenes debe ser el mismo tanto para la versión de GrPPI como para el resto de las versiones.*	
Estabilidad	*Estable*		
Tipo	*Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.6: *Requisito Funcional FR_05*

ID	*FR_06*	**Requisito**	*Resultado Fluidanimate*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*La velocidad y posición final de las partículas obtenidas con la versión implementada usando GrPPI debe ser igual a los obtenidos con el resto de las versiones.*	
Estabilidad	*Estable*		
Tipo	*Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.7: *Requisito Funcional FR_06*

ID	*FR_07*	**Requisito**	*Resultado Raytrace*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*El modelo de rayos generado por la versión de GrPPI debe ser igual al obtenido con el resto de las implementaciones.*	
Estabilidad	*Estable*		
Tipo	*Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.8: *Requisito Funcional FR_07*

ID	*FR_08*	**Requisito**	*Resultado Streamcluster*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*Los datos obtenidos para los centros (id, peso y coordenadas) usando la versión implementada con GrPPI debe coincidir con los obtenidos con el resto de las versiones.*	
Estabilidad	*Estable*		
Tipo	*Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.9: *Requisito Funcional FR_08*

ID	*FR_09*	**Requisito**	*Resultado Streamcluster*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*Las aplicaciones programadas usando GrPPI se deben ejecutar usando los mismos comandos que se usan en la ejecución del resto de las versiones.*	
Estabilidad	*Estable*		
Tipo	*Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.10: *Requisito Funcional FR_09*

Como se puede ver hay un requisito funcional por cada una de las aplicaciones del *benchmark* que se van a implementar, esto es debido a que cada una pertenece a un dominio diferente y por tanto los resultados que se deben obtener son distintos para cada una de ellas.

3.1.2. Requisitos No Funcionales

Los requisitos no funcionales son aquellos que en lugar de funciones establecen las restricciones que debe tener la implementación. En otras palabras, los requisitos no funcionales son los que establecen los atributos de calidad gracias a los cuales se puede evaluar la implementación. Dentro de este tipo de requisitos se encuentran los relacionados con la seguridad del sistema, la usabilidad, el rendimiento, escalabilidad, mantenibilidad, tiempo de respuesta, etc.

A continuación, se muestran los requisitos no funcionales que se han establecido para este proyecto:

ID	*NFR_01*	**Requisito**	*Sistema Operativo*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*Las aplicaciones desarrolladas y evaluadas deben ser compatibles con la versión 14.04.2 LTS de Ubuntu*	
Estabilidad	*No Estable*		
Tipo	*No Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.11: *Requisito No Funcional NFR_01*

ID	*NFR_02*	**Requisito**	*Compilador*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*Las aplicaciones deben compilar al usar la versión 6.0 o superior del compilador gcc*	
Estabilidad	*No Estable*		
Tipo	*No Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.12: *Requisito No Funcional NFR_02*

ID	*NFR_03*	**Requisito**	*Lenguaje de Programación*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*Las aplicaciones se deben implementar utilizando el lenguaje de programación C++*	
Estabilidad	*Estable*		
Tipo	*No Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.13: *Requisito No Funcional NFR_03*

ID	*NFR_04*	**Requisito**	*Modelo de Programación*
Necesidad	*Necesario*	**Texto del Requisito**	
Prioridad	*Alta*	*Las aplicaciones se deben implementar utilizando la biblioteca de patrones de diseño paralelos GrPPI*	
Estabilidad	*Estable*		
Tipo	*No Funcional*		
Estado	*Validado*		
Verificabilidad	*Verificable*		

Tabla 3.14: *Requisito No Funcional NFR_04*

3.2. Análisis de los Requisitos

Una vez se han expuesto los requisitos que deben cumplir las aplicaciones, se puede pasar a analizar cuáles de ellos se han cumplido y cuáles no, para ello se han realizado una serie de casos de uso que corresponden con las aplicaciones que se han implementado. Los resultados de dicho análisis se van a mostrar en dos matrices de trazabilidad: una para los requisitos funcionales y otra para los requisitos no funcionales.

A continuación, se muestra la matriz relativa a los requisitos funcionales expuestos en la Sección 3.1.1:

Casos de Uso	Requisitos Funcionales								
	FR_01	FR_02	FR_03	FR_04	FR_05	FR_06	FR_07	FR_08	FR_09
Blackscholes	✓								✓
Bodytrack		✓							✓
Canneal			✓						✓
Facesim				✓					✓
Ferret					✓				✓
Fluidanimate						✓			✓
Raytrace							✓		✓
Streamcluster								✓	✓

Tabla 3.15: *Matriz de trazabilidad de los requisitos funcionales con los casos de uso realizados*

Como se puede ver los requisitos funcionales concuerdan con casos de uso, esto es debido a que cada uno de los estos requisitos hace referencia a la salida especifica de una de las aplicaciones.

En cuanto a los requisitos no funcionales, al ser características comunes a todas las aplicaciones o casos de uso, se deben cumplir en todos los casos, como se puede ver en la matriz inferior:

Casos de Uso	Requisitos No Funcionales			
	NFR_01	NFR_02	NFR_03	NFR_04
Blackscholes	✓	✓	✓	✓
Bodytrack	✓	✓	✓	✓
Canneal	✓	✓	✓	✓
Facesim	✓	✓	✓	✓
Ferret	✓	✓	✓	✓
Fluidanimate	✓	✓	✓	✓
Raytrace	✓	✓	✓	✓
Streamcluster	✓	✓	✓	✓

Tabla 3.16: *Matriz de trazabilidad de los requisitos no funcionales con los casos de uso realizados*

Además, cabe decir que como se puede observar en ambas matrices todos los requisitos tanto funcionales como no funcionales están cubiertos al menos por un caso de uso.

Capítulo 4

Diseño e Implementación

En este capítulo se explica con más detalle el modelo de programación paralela *GrPPI* que ha sido el utilizado para la implementación (Sección 4.1), así como cada una de las aplicaciones escogidas dentro del *benchmark* P^3ARSEC para realizar la evaluación (Sección 4.2). En cuanto a las aplicaciones del *benchmark* P^3ARSEC, se describen tanto las aplicaciones en sí, como la implementación secuencial desarrollada por la Universidad de Princeton, a partir de la cual se identifican las partes potencialmente paralelizables. Por último, se describe el modo en el que han sido paralelizadas con los modelos ya disponibles en el *benchmark* y el procedimiento seguido para su paralelización usando *GrPPI*.

4.1. GrPPI

Esta sección es una continuación de la Sección 2.2.6 en la que se describió en términos generales en que consiste este modelo de programación, de forma que a continuación simplemente se detalla el funcionamiento de este.

GrPPI, como ya se ha comentado, permite evaluar cuál de los modelos de programación paralela de los dispone es el más adecuado en cada caso sin necesidad de modificar el código, esto es debido a que posee la implementación interna de varios modelos de programación que se pueden usar durante la ejecución.

Los modelos de programación para los que consta actualmente GrPPI, a parte del modo secuencial, son: *OpenMP, TBB, ISO C++ Threads y FastFlow*, aunque para la evaluación solo se han utilizado *OpenMP, TBB e ISO C++ Threads*, ya que cuando se inició este proyecto GrPPI solo tenía implementación para esos tres. Para cambiar entre un modelo de programación y otro simplemente se debe cambiar el modo de ejecución en los patrones utilizados. A continuación, se muestra la forma de declarar cada uno de los modos de ejecución que se han usado:

```
//GrPPI_Native
    grppi::parallel_execution_native e_native;
//GrPPI_OpenMP
    grppi::parallel_execution_omp e_openmp;
//GrPPI_TBB
    grppi::parallel_execution_tbb e_tbb;
```

Listado 4.1: *Formato para la declaración de los modos de ejecución (modelos de programación).*

En este modelo los patrones de paralelismo se pueden clasificar en: patrones paralelos de datos como es el caso de los patrones *Map, Reduce, MapReduce, Parallel_for y Stencil*, patrones paralelos de tareas como el patrón *Divide & Conquer* y patrones de streaming como es el caso de los patrones *Pipeline, Farm, Stream filter, Stream reduction y Stream iteration*. Para poder usar cada uno de los patrones ya mencionados se tiene que usar un formato fijo, el cual se muestra a continuación:

- Map

```
grppi::map(modo_ejecucion, begin(vec_entrada), end(vec_entrada), begin(vec_salida),
    [](auto parametro) {
        //Computo que se quiere realizar
        return resultado;
    }
);
```

Listado 4.2: *Formato para la implementación del patrón Map usando GrPPI.*

- Reduce

```
grppi::reduce(modo_ejecucion, begin(vec_entrada), end(vec_entrada), valor_inicial,
    [](auto parametro1, auto parametro2) {
        //Computo que se quiere realizar
        return resultado;
    }
);
```

Listado 4.3: *Formato para la implementación del patrón Reduce usando GrPPI.*

- MapReduce

```
grppi::map_reduce(modo_ejecucion, begin(vec_entrada), end(vec_entrada), valor_inicial,
    [](auto parametro_map) {
        //Computo que se quiere realizar en el Map
        return resultado_map ;
    }, [] (auto parametro1_reduce, auto parametro2_reduce) {
        //Computo que se quiere realizar en el Reduce
        return resultado_reduce;
    }
);
```

Listado 4.4: *Formato para la implementación del patrón MapReduce usando GrPPI.*

- Parallel_for

```
grppi::parallel_for(modo_ejecucion, inicio, fin, step, chunk_size,
    [](auto parametros) {
        //Computo que se quiere realizar
    }
);
```

Listado 4.5: *Formato para la implementación del patrón Parallel_for usando GrPPI.*

- Stencil

```
grppi::stencil(modo_ejecucion, begin(vec_entrada), end(vec_entrada), begin(vec_salida),
    [](auto iterador, auto vecindario) {
        //Computo que se quiere realizar
        return resultado;
    }, [] (auto iterador1, auto iterador2){
        //Definicion del vecindario que se va a usar en el computo
        return vecinos;
    }
);
```

Listado 4.6: *Formato para la implementación del patrón Stencil usando GrPPI.*

- Divide & Conquer

```
grppi::divide_conquer(modo_ejecucion, begin(vec_entrada), end(vec_entrada),
    [](auto parametros_dividir) {
        //Computo para dividir el problema en subproblemas
        return resultado_division;
    }, [] (auto parametros_predicado){
        //Predicado que determina si el problema es elemental y se puede resolver o no
        return resultado_condicion;
    }, [] (auto parametros_resolver){
        //Computo para resolver cada uno de los problemas elementales
        return resultado_parcial;
    }, [] (auto parametros_combinar){
        //Codigo para combinar los resultados obtenidos
        return resultado_final;
    }
);
```

Listado 4.7: *Formato para la implementación del patrón Divide & Conquer usando GrPPI.*

- Pipeline

```
grppi::pipeline(modo_ejecucion,
    []() -> optional<...> {
        //Computo para generar los datos que se van a pasar a la siguiente etapa
        return datos;
    }, [] (auto parametros_etapa_intermedia){
        //Computo que se quiere realizar despues de generar los datos
        return resultado_etapa;
    }, ... //Puede haber tantas etapas intermedias como se quiera
    [] (auto parametros_consumidor){
        //Computo que se quiere realizar al final con los datos obtenidos de las etapas
            de computo del Pipeline
    }
);
```

Listado 4.8: *Formato para la implementación del patrón Pipeline usando GrPPI.*

- Farm

```
grppi::pipeline(modo_ejecucion,
    []() -> optional<...> {
        //Computo para generar los datos que se van a pasar a la siguiente etapa
        return datos;
    },
    grppi::farm (numero_hilos,
        [] (auto parametros_etapa_intermedia){
        //Computo que se quiere realizar en paralelo con el numero de hilos especificado
             con los datos generados
        return resultado_etapa_intermedia;
    }), ... //Puede haber tantas etapas intermedias como se quiera
    [] (auto parametros_consumidor){
        //Computo que se quiere realizar al final con los datos obtenidos de las etapas
            de computo del Pipeline
    }
);
```

Listado 4.9: *Formato para la implementación del patrón Farm usando GrPPI.*

- Stream Filter

```
grppi::pipeline(modo_ejecucion,
    []() -> optional<...> {
        //Computo para generar los datos que se van a pasar a la siguiente etapa
        return datos;
    },
    grppi::keep (
        [] (auto parametros_predicado){
        //Codigo para seleccionar aquellos datos que se mantienen y por tanto, pasan al
            consumidor
        return datos_a_mantener;
     }), //Se puede usar tanto el keep como el discard, en este ejemplo simplemente se
         han puesto ambos para que se viera el formato
    grppi::discard (
        [] (auto parametros_predicado){
        //Codigo para seleccionar aquellos datos que se descartan y por tanto, no pasan
```

```
            al consumidor
        return datos_a_descartar ;
    }),
    [] (auto parametros_consumidor){
        //Computo que se quiere realizar al final con los datos obtenidos de las etapas
            de computo del Pipeline
    }
);
```

Listado 4.10: *Formato para la implementación del patrón Stream Filter usando GrPPI.*

- Stream Reduction

```
grppi::pipeline(modo_ejecucion, []() -> optional<...> {
        //Computo para generar los datos que se van a pasar a la siguiente etapa
        return datos;
    },
    grppi::stream_reduce (magnitud_ventana, desplazamiento, valor_identidad,
        [] (auto parametros_reduccion){
        //Computo que se quiere realizar para combinar los valores de la entrada
        return resultado_reduccion;
    }), ... //Puede haber tantas etapas intermedias como se quiera
    [] (auto parametros_consumidor){
        //Computo que se quiere realizar al final con los datos obtenidos de las etapas
            de computo del Pipeline
    }
);
```

Listado 4.11: *Formato para la implementación del patrón Stream Reduction usando GrPPI.*

- Stream Iteration

```
grppi::pipeline(modo_ejecucion, []() -> optional<...> {
        //Computo para generar los datos que se van a pasar a la siguiente etapa
        return datos;
    },
    grppi::repeat_until ( [] (auto parametros_computo){
        //Computo que se quiere realizar
        return resultado;
```

```
        }, [] (auto parametros_predicado){
        //Predicado para determinar el final del bucle
        return resultado_condicion;
    }), ... //Puede haber tantas etapas intermedias como se quiera
    [] (auto parametros_consumidor){
        //Computo que se quiere realizar al final con los datos obtenidos de las etapas
            de computo del Pipeline
    }
);
```

Listado 4.12: *Formato para la implementación del patrón Stream Iteration usando GrPPI.*

Algunos de estos patrones que se acaban de explicar tienen características adicionales: la composición, hay patrones que se pueden componer con otros como es el caso de los *pipelines* que se pueden combinar con cualquier otro patrón, mayor número de entradas, algunos patrones como el *map* aceptan más de una entrada, etc. Para una información más amplia de cada uno de los patrones se puede consultar la documentación [31].

4.2. P^3ARSEC

Como se indica en la Sección 2.3 hay una gran variedad de aplicaciones disponibles en este *benchmark*, pero la aplicación no es lo único que se puede cambiar para obtener diversidad en los resultados, ya que cada uno de esos programas tienen 6 conjuntos de entrada distintos con los que se pueden ejecutar para obtener distintos resultados.

- **test:** es un conjunto de entrada muy pequeño que se utiliza simplemente para comprobar la funcionalidad básica de la aplicación.
- **simdev:** es un conjunto de entrada pequeño al igual que el anterior, pero en este caso está destinado a las pruebas de simuladores en desarrollo ya que garantiza un comportamiento básico similar al que tendría un programa real.
- **simsmall, simmedium y simlarge:** estos tres conjuntos de entrada son ideales para realizar estudios de microarquitecturas con simuladores. Estos conjuntos son aproximaciones un poco más toscas que sacrifican la precisión por la tratabilidad.
- **native:** este último conjunto es el más grande de los anteriores y está destinado a la realización de estudios de rendimiento en máquinas reales, puesto que se exceden las demandas computacionales frecuentes en dicho dominio de aplicación.

Por otro lado, en cuanto a las aplicaciones seleccionadas, cabe decir que a pesar de que el *benchmark* consta de 12 aplicaciones, no se ha realizado la evaluación del rendimiento de todas ellas. A continuación, se explica brevemente en que consiste cada uno de los programas que se han escogido para realizar dicha evaluación, tal como se especifican en los artículos publicados por la Universidad de Princeton [27, 28, 32].

4.2.1. Blackscholes

Blackscholes es una aplicación del benchmark Intel RMS usada en el ámbito de análisis financiero, que calcula los precios de una cartera de opciones europeas mediante el uso de la ecuación diferencial de Black-Scholes, que es la siguiente:

$$\frac{\partial V}{\partial T} + \frac{1}{2}\sigma^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{4.1}$$

En la ecuación 4.1 la V representa el precio de una opción en función del precio de la acción S y del tiempo que queda hasta la expiración de dicha opción T, r es la tasa de interés libre de riesgo, y σ es la volatilidad de dicha acción. Esta ecuación permite asegurar la acción comprando y vendiendo el activo de la manera correcta, lo que supone la eliminación del riesgo que conllevan las inversiones en bolsa.
La ecuación de Black-Scholes permite operaciones de compra *(Call)* si se usa la ecuación 4.2 o de venta *(Put)* si se usa la ecuación 4.3 [33].

$$C = SN(d1) - Ve^{rT}N(d2) \tag{4.2}$$

$$P = Ve^{-rT}N(-d2) - SN(-d1) \tag{4.3}$$

donde:

$$d1 = \frac{ln(\frac{S}{V}) + (r + \frac{\sigma^2}{2})T}{\sigma^2\sqrt{T}} \tag{4.4}$$

$$d2 = d1 - \sigma^2\sqrt{T} \tag{4.5}$$

Esta aplicación consta de un archivo de datos de entrada, a través del cual se obtienen los valores de las opciones que se almacenan en la estructura OptionData. Una vez se tiene la cartera de opciones, el programa itera entre las opciones y llama a la función BlkSchlsEqEuroNoDiv para calcular su precio.
Finalmente, una vez que se han calculado los precios mediante el uso de la ecuación adecuada (4.2 o 4.3), todos los valores resultantes se escriben en un fichero de salida, el cual constituye la salida del propio programa.

A continuación, se muestra una tabla en la que se exponen los modelos de programación para los cuales hay una implementación de la aplicación Blackscholes:

Programa	Modelo de Programación						
	Pthreads	OpenMP	TBB	FastFlow	GrPPI_Native	GrPPI_OpenMP	GrPPI_TBB
BlackScholes	✓	✓	✓	✓	✓	✓	✓

Tabla 4.1: *Modelos de programación paralela para los que tiene implementación la aplicación BlackScholes*

En cuanto a los archivos de código para esta aplicación, cabe decir que cada una de las implementaciones constituye un único archivo: *blackscholes.c* en el caso de la implementación secuencial, *Pthreads, OpenMP, TBB y FastFlow* y *blackscholes_grppi.cpp* en el caso de la implementación usando *GrPPI*.

Implementación

Una vez descrita la aplicación Blackscholes, se va a exponer la implementación secuencial propuesta en el benchmark PARSEC para dicha aplicación, así como mencionar las partes potencialmente paralelizables del programa.

El programa consta de dos funciones comunes (*CNDF y BlkSchlsEqEuroNoDiv*) y un *main* principal y un *main* individual para cada una de las implementaciones. La función *CNDF* (Cumulative Normal Distribution Function) es la encargada de calcular los valores resultantes de las Ecuaciones 4.4 y 4.5, mientras que la función *BlkSchlsEqEuroNoDiv* coge los resultados de la función anterior y aplica la Ecuación 4.3 en el caso de realizar una venta o la Ecuación 4.2 en el caso de realizar una compra, obteniendo así los precios finales para las opciones.

El *main* principal es el encargado de coger los datos de las opciones de la cartera necesarios para aplicar la ecuación de Blackscholes y guardarlos en estructuras del tipo OptionData. Una vez que los datos están guardados llama al *main* individual correspondiente dependiendo de la implementación que se quiera ejecutar y por último cuando ya se han obtenido los resultados los escribe en el fichero de salida.

Por último, el *main* individual para cada implementación es el que itera por las distintas opciones y llama a la función *BlkSchlsEqEuroNoDiv*. Además, antes de volver al *main* principal para guardar los valores comprueba que no haya habido ningún error, calculando el precio delta y comprobando que no fuera menor que 10^{-4}.

La única parte que se puede paralelizar es el hecho de calcular los precios para las opciones de forma simultánea. A continuación, se expone la manera en la que se ha implementado dicha paralelización con cada uno de los modelos de programación de los que ya constaba el *benchmark* P^3ARSEC:

- **Pthreads**: Se crean tantos hilos como se especifican con el comando de ejecución y en función del número de hilos se divide la cartera de opciones para asignar la misma cantidad de opciones a cada uno de los hilos, salvo en el caso de que esa división no sea exacta, lo que provocaría que uno de los hilos tuviera más opciones que el resto. Luego cada uno de esos hilos itera sobre sus opciones y va haciendo los cálculos tal como se ha comentado anteriormente.

- **OpenMP**: Se usa la directiva *#pragma omp parallel for*, estableciendo ciertas variables como privadas para evitar las condiciones de carrera.

- **TBB**: Se usan las estructuras *tbb::split*, *tbb::blocked_range<int>*, *tbb::affinity_partitioner* y *tbb::parallel_for*.

- **FastFlow**: Se usa el patrón de paralelismo *parallel_for* y en concreto el *parallel_for_thid*, ya que usan el identificador de los hilos para asignar a cada uno de ellos una porción concreta de las opciones.

Implementación con GrPPI

En el caso de *GrPPI* se han desarrollado varias versiones: una en la que se usa el patrón paralelo *parallel_for* y otra en la que se usan dos patrones anidados, un *farm* con un *map* en su interior. La implementación de ambas versiones se puede ver a continuación:

```
for (j=0; j<NUM_RUNS; j++) {
grppi::parallel_for(e, 0, numOptions, 1, 500,
         [&](int i) {
           fptype price;
           fptype priceDelta;

           price = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i],
                                        rate[i], volatility[i], otime[i], otype[i], 0);

           priceDelta = data[i].DGrefval - price;
           if( fabs(priceDelta) >= 1e-4 ){
               printf("Error on %d. Computed=%.5f, Ref=%.5f, Delta=%.5f\n",
                     i, price, data[i].DGrefval, priceDelta);
               numError ++;
           }
           prices[i] = price;
         }
       );
}
```

Listado 4.13: *Código del patrón parallel_for (primera versión de la implementación de la aplicación Blackcholes).*

```
grppi::pipeline(e,
    [&]() mutable -> optional<int> {
      if (j<=NUM_RUNS) return j++;
      else return {};
    },
    grppi::farm(n,
      [&](int x) {
        grppi::map(e, begin(vec), end(vec), begin(prices),
          [&](int i) {
            fptype price;
            fptype priceDelta;

            price = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i],
                                         rate[i], volatility[i], otime[i], otype[i], 0);

            priceDelta = data[i].DGrefval - price;
            if( fabs(priceDelta) >= 1e-4 ){
                printf("Error on %d. Computed=%.5f, Ref=%.5f, Delta=%.5f\n",
                       i, price, data[i].DGrefval, priceDelta);
                numError ++;
            }
          return price;
          }
        );
      return x;
      }
    ),
    [](int x) { ; }
  );
```

Listado 4.14: *Código de los patrones anidados farm y map (segunda versión de la implementación de la aplicación Blackcholes).*

Las dos versiones del código podrían ser buenas, pero a simple vista no es posible determinar cuál de ellas supondrá mayor aumento en la eficiencia del programa, por ello se ha decidido evaluar ambas (Sección 5.3.1) y determinar una vez hecha la evaluación cuál es mejor.

4.2.2. Bodytrack

Bodytrack es una aplicación del ámbito de visión por ordenador y reconocimiento de patrones cuyo objetivo es monitorizar la pose en 3D de un cuerpo humano a partir de una secuencia de imágenes tomadas de manera simultánea usando múltiples cámaras. Para realizar un seguimiento de las posturas en las imágenes, se establecen como características principales de estas los bordes y la silueta en primer plano y se emplea un filtro de partículas, que se basa en un modelo de cuerpo de árbol cinemático 3D compuesto por segmentos. De esta forma se evita tener que depender de suposiciones tales como la existencia de marcadores o movimientos restringidos.

En cuanto al árbol cinemático, cabe destacar que cada uno de los segmentos se representa mediante un cilindro cónico que simboliza cada una de las partes del cuerpo humano. En concreto se usan 10 segmentos: dos en cada una de las extremidades, uno para el torso y otro para la cabeza, como se puede ver en la Figura 4-1. Cada uno de estos cilindros viene representado por su radio y longitud, y se unen en el árbol cinemático basándose en los ángulos de las uniones.

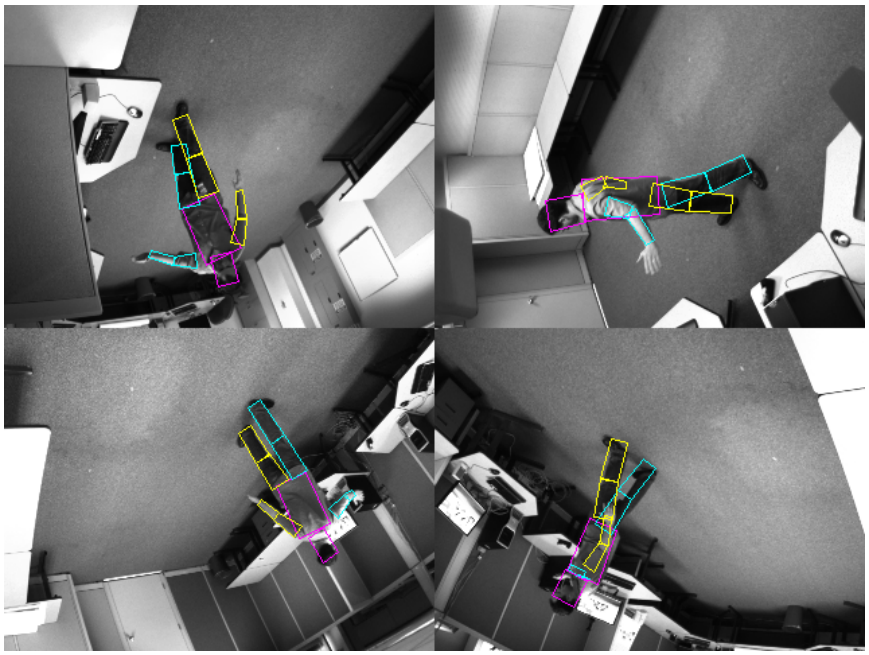


Figura 4-1: *Representación del árbol cinemático sobre un conjunto de imágenes*

Cada una de las partículas del filtro simboliza tanto el conjunto de uniones de los distintos segmentos como una translación global y vienen representadas por una probabilidad, que es la medida de alineación del modelo de cuerpo en 3D con la silueta en primer plano y los bordes de la imagen. Para determinar el valor de dicha probabilidad primero se genera el modelo de cuerpo en un espacio 3D y luego se proyecta como un cuadrilátero en cada una de las imágenes 2D que tiene como entrada la aplicación. Esto permite calcular el valor de verosimilitud basándose en las características de la imagen ya mencionadas (bordes y

silueta), tomando muestras dentro del espacio 2D que se comparan con la silueta binaria en primer plano. Aquellas muestras que concuerden con el primer plano y por tanto estén más cercanas a los bordes de la imagen contribuyen positivamente en la probabilidad, mientras que las muestras que coinciden con el fondo suponen una penalización para dicha probabilidad.

Esta aplicación consta de varias imágenes de entrada codificadas en binario, de forma que cada píxel de la imagen sería un conjunto de *bits* distinto. Estos bits se analizan mediante una máscara basada en el gradiente para determinar cuáles de ellos son los podrían ser parte del borde y una vez obtenidos los resultados se comparan con un umbral para descartar los falsos positivos. Una vez se han descubierto los bordes, se pasa a aplicar un filtro Gaussiano que asigna a cada uno de los píxeles un valor entre 0 y 1 en función de la distancia entre dicho píxel y el borde de la imagen. Y, por último, se calculan los valores de las partículas usando los resultados de las dos operaciones anteriores.

A continuación, se muestra una tabla en la que se exponen los modelos de programación paralela para los cuales la aplicación Bodytrack tiene implementación:

Programa	Modelo de Programación						
	Pthreads	OpenMP	TBB	FastFlow	GrPPI_Native	GrPPI_OpenMP	GrPPI_TBB
Bodytrack	✓	✓	✓	✓	✓	✓	✗

Tabla 4.2: *Modelos de programación paralela para los que tiene implementación la aplicación Bodytrack*

En el caso de la aplicación Bodytrack, hay varios archivos que son comunes a todos los modelos de programación, el *main.cpp* y luego cada uno de ellos tiene tres archivos de código propios en los que se implementan las funciones, que en el caso de *GrPPI* son: *TrackingModelGRPPI.cpp, TrackingModelGRPPI.h y ParticleFilterGRPPI.h*, siendo los archivos correspondientes a la versión serial: *TrackingModel.cpp, TrackingModel.h y ParticleFilter.h.*

Implementación

Esta aplicación tiene como entrada un conjunto de imágenes extraídas de distintas cámaras y una serie de parámetros como el número de cámaras, el número de *frames* que se deben procesar, el número de partículas, el número de capas, el modelo con el que se desea ejecutar la aplicación, el número de hilos y opcionalmente el nombre del archivo de salida (*.bmp*).

Dependiendo del modelo de ejecución establecido el *main* llama a una función u otra (*mainGRPPI, mainTBB, mainFF...*), en la que se crean dos objetos, uno del modelo adecuado (*TackingModelGRPPI, TrackingModelTBB, TrackingModelFF...*) y otro del filtro de partículas (*ParticleFilterGRPPI, ParticleFilterTBB, ParticleFilterFF...*) a partir de los cuales se accederá a los tres archivos de código específicos para

cada modelo.

Después se cargan y procesan todas las imágenes para realizar una observación en el intervalo de tiempo 0, mediante la función *GetObservation*. Por cada una de las imágenes se crea un mapa de bordes, calculando la magnitud umbral del gradiente y aplicando un filtro Gaussiano, esto permite descartar del conjunto de bordes potenciales los falsos positivos.

Una vez hecho eso, se genera un conjunto inicial de partículas y luego se itera tantas veces como el número de *frames* establecido en la entrada, estimando el valor del conjunto de partículas, escribiendo el resultado en el archivo *poses.txt* y el modelo en el archivo de salida establecido en la entrada (*.bmp*).

Las partes potencialmente paralelas en esta aplicación son: el cálculo de la magnitud umbral del gradiente, la aplicación del filtro Gaussiano y el cálculo de los pesos de las partículas al generar un conjunto inicial. A continuación, se describe la manera en la que se ha implementado dicho paralelismo para cada uno de los modelos de programación de los que ya constaba el *benchmark* P^3ARSEC:

- **Pthreads**: Se usa un *pool* de hilos implementada en un archivo adicional denominado *WorkPoolThread.h*.
- **OpenMP**: Se paralelizan los bucles utilizando la primitiva *#pragma omp parallel for*.
- **TBB**: El paralelismo viene determinado por la sobrecarga del operador () y el uso de las funciones *tbb::blocked_range* y *tbb:parallel_for*.
- **FastFlow**: Se reescriben los bucles para que coincidan con el patrón *Parallel_for*.

Implementación con GrPPI

En el caso de *GrPPI*, se ha intentado hacer una implementación equivalente a la de *FastFlow*, por lo que se ha usado el patrón *Parallel_for*. Como esta aplicación no consta de un único patrón, sino que el patrón *Parallel_for* se usa cinco veces dentro de la implementación, a continuación solo se muestra un ejemplo del patrón en el código y no todo este.

```
grppi::parallel_for(e, 1, (src.Height()-1), 1, CHUNKSIZE,
   [&](auto y) {
               Im8u *p = &src(1,y), *ph = &src(1,y - 1), *pl = &src(1,y + 1),
                    *pr = &r(1,y);
               for(int x = 1; x < src.Width() - 1; x++){
                   float xg = -0.125f * ph[-1] + 0.125f * ph[1] - 0.250f * p[-1] + 0.250f
                        * p[1] - 0.125f * pl[-1] + 0.125f * pl[1];
```

```
                    float yg = -0.125f * ph[-1] - 0.250f * ph[0] - 0.125f * ph[1] +
                        0.125f * pl[-1] + 0.250f * pl[0] + 0.125f * pl[1];
                    float mag = xg * xg + yg * yg;
                    *pr = (mag < threshold) ? 0 : 255;
                    p++; ph++; pl++; pr++;
                }
  }
);
```

Listado 4.15: *Código de implementación de uno de los patrones parallel_for de la aplicación Bodytrack.*

4.2.3. Canneal

Canneal es un kernel desarrollado por la Universidad de Princeton para el ámbito de ingeniería que utiliza el método SA (*Simulated Annealing*) con reconocimiento de caché, lo que permite reducir el coste del enrutamiento en el diseño de *chips*.

Canneal elige de forma pseudoaleatoria pares de elementos *netlist* y trata de intercambiarlos mediante un *swap*. Esa elección pseudoaleatoria se realiza mediante la obtención de un número aleatorio empleando el generador *Mersenne Twister*. En cada iteración se descarta un solo elemento para intentar aumentar la reutilización de los datos, lo que provoca una reducción en la capacidad de fallos de la caché *(cache misses)*. El algoritmo SA utilizado favorece los intercambios que disminuyen el coste del enrutamiento, aunque con cierta probabilidad también permite que se produzcan intercambios desventajosos, es decir, que provoquen un aumento del coste de enrutamiento, ya que esto permite escapar de los mínimos locales. Esta probabilidad va disminuyendo a medida que pasa el tiempo para permitir que el diseño finalmente converja.

Este programa se basa en tratar de recuperarse de las condiciones de carrera entre los datos en lugar de evitarlas, lo que constituye una estrategia de sincronización agresiva. Para realizar los intercambios, los punteros a los distintos elementos *netlist* son desreferenciados e intercambiados de forma automática. Durante la evaluación del intercambio no se mantienen los bloqueos *(block)*, lo que puede desembocar en intercambios desventajosos y provoca un aumento de la probabilidad de aceptar intercambios que aumenten el coste de enrutamiento para favorecer la recuperación automática del método SA. Para sincronizar los intercambios realizados sin usar bloqueos se utiliza una implementación basada en instrucciones atómicas.

A continuación, se muestra una tabla en la que se exponen los distintos modelos de programación paralela para los cuales el *kernel Canneal* posee implementación:

Programa	Modelo de Programación						
	Pthreads	OpenMP	TBB	FastFlow	GrPPI_Native	GrPPI_OpenMP	GrPPI_TBB
Canneal	✓	✗	✗	✓	✓	✓	✓

Tabla 4.3: *Modelos de programación paralela para los que tiene implementación la aplicación Canneal*

Este programa consta de varios archivos de código, aunque los principales son *main.cpp* y *annealer_thread.cpp*, que serán los que contengan tanto el código secuencial como el paralelo, salvo en el caso del modelo de programación paralela *FastFlow* que implementa su propia versión de la clase *annealer_thread.cpp*, *annealer_thread_ff.cpp*.

Implementación

El *kernel Canneal* tiene como entradas el número de intercambios por unidad de temperatura, la temperatura inicial y el nombre del archivo que contiene los elementos de tipo *netlist* que se van a utilizar. Además de estos parámetros que son fijos el programa acepta de forma opcional un argumento adicional relativo al número de pasos que se quieren realizar, en el caso de que este parámetro no se especifique significará que el programa continuará la ejecución hasta que el diseño converja.

En primer lugar, se crean tanto hilos como se haya especificado en la entrada (en el caso de la versión secuencial solo 1) y se enlazan con la función *entry_pt*, la cual crea punteros de tipo *annealer_thread*. La clase *annealer_thread* consta de un método principal, el *Run* que concentra todo el computo del programa.

El método *Run* lo primero que hace es escoger de forma pseudoaleatoria dos elementos de tipo *netlist* mediante el uso de la función *get_random_element*. Una vez seleccionados, se realizan tantas iteraciones como el número de pasos especificado o hasta que el diseño haya convergido, y para cada una de esas iteraciones se realizaran tantos intercambios como se indique en la entrada. Para cada uno de estos intercambios, se calcula la variación que se produce en el coste total de enrutamiento, mediante la función *calculate_delta_routing_cost* y se evalúa en cada caso si ese intercambio se acepta o no, mediante el cálculo del coste que conlleva el intercambio que se quiere realizar y la temperatura actual (función *accept_move*). En el caso de que se acepte el intercambio independientemente de si este el bueno o no, se llama a la función *swap_locations* y si se rechaza simplemente se pasa al siguiente intercambio.

El resultado de este programa es el coste de enrutamiento final que se obtiene una vez se han finalizado todas las iteraciones necesarias.

La paralelización en este *kernel* se encuentra en el número de hilos que se crean, así como en la forma de creación y ejecución de dichos hilos. A continuación, se expone la manera en la que se ha implementado dicha paralelización con cada uno de los modelos de programación de los que ya constaba el *benchmark*

P^3*ARSEC*:

- **Pthreads**: En este caso los hilos se crean mediante el uso de la función *pthread_create*. Todos los hilos creados se sincronizan posteriormente mediante el uso de la función *pthread_join*.
- **FastFlow**: En este caso esos hilos se crean como trabajadores del patrón *Farm*.

Implementación con GrPPI

En el caso de *GrPPI* en lugar de utilizar el patrón *Farm* como en *FastFlow*, se ha optado por el uso del patrón *Parallel_for*, cuya implementación se puede ver a continuación:

```
void* thread_in = static_cast<void*>(&a_thread);
grppi::parallel_for(e, 0, num_threads, 1, 1,
    [&](auto worker){
        entry_pt(thread_in);
    }
);
```

Listado 4.16: *Código de implementación del patrón parallel_for para el kernel Canneal.*

4.2.4. Facesim

Facesim es una aplicación del benchmark Intel RMS desarrollada por la Universidad de Stanford para el ámbito de la animación. Esta toma un modelo de una cara humana y trata de reproducirla de forma realista mediante la simulación de la física subyacente (la secuencia de activaciones musculares).

Para ello emplea un modelo 3D compuesto por una malla de partículas (tantas como se especifique en la entrada) que representa el rostro y dos superficies triangulares para representar el cráneo y la mandíbula. Dicha malla se divide de forma estática para permitir la paralelización del programa, formando particiones. Esas partículas se dividen a su vez en nodos que pueden pertenecer a una o más particiones. Si los datos se extienden sobre un nodo que pertenece a más de una partición, esos datos se deben replicar para cada una de esas divisiones.

Las fuerzas físicas y el movimiento del modelo se van calculando *frame a frame*, permitiendo de esta forma producir la animación deseada. Los movimientos que se producen en el modelo son debidos a las variaciones en las posiciones de cada una de las partículas de la malla, las cuales se obtienen mediante el uso del método de Newton para la resolución de sistemas de ecuaciones.

A continuación, se muestra una tabla en la que se establecen los modelos de programación paralela para los cuales la aplicación Facesim posee implementación:

Programa	Modelo de Programación						
	Pthreads	OpenMP	TBB	FastFlow	GrPPI_Native	GrPPI_OpenMP	GrPPI_TBB
Facesim	✓	✗	✗	✓	✓	✓	✓

Tabla 4.4: *Modelos de programación paralela para los que tiene implementación la aplicación Facesim*

Esta aplicación consta de una gran variedad de archivos de código, pero en concreto los que se modifican para llevar a cabo el paralelismo son: *FACE_EXAMPLE.h, ARRAY_PARRALLEL_OPERATIONS.cpp, DIAGONALIZED_FINITE_VOLUME_3D.cpp y DEFORMABLE_OBJECT.cpp*, todos ellos comunes al conjunto de modelos de programación paralela.

Implementación

Una de las entradas a esta aplicación es un conjunto de archivos en los que se especifica el número de partículas que debe tener la malla, sus posiciones, velocidades y las fuerzas que actúan sobre ellas. A parte de esos archivos se le proporciona al programa el número de tetraedros y el número de *frames* que se quieren usar.

Esas partículas que forman la malla, como ya se ha comentado, se agrupan en particiones que se pueden ejecutar en paralelo. En cada unidad de tiempo se procesan todos los elementos con uno o más nodos propiedad de la partícula, aunque finalmente solo se mantengan los resultados de los nodos pertenecientes a la partición. Antes de finalizar cada iteración se calcula el estado de la malla usando la función *Advance_One_Time_Step_Quasistatic*.

Dicha función lleva a cabo tres procesos: actualizar el estado de la malla, agregar las fuerzas que actúan sobre la malla y resolver el sistema de ecuaciones. Para actualizar el estado utiliza el método de Newton ya mencionado, el cual consiste en reducir un sistema de ecuaciones no lineal en otro lineal equivalente determinado y simétrico, para que se pueda solucionar posteriormente usando un solucionador de gradiente conjugado.

Con el fin de agregar las fuerzas que actúan sobre la malla, primero es necesario calcularlas. Una vez se han obtenido las fuerzas se extraen las posiciones de los vértices de los tetraedros y se evalúa la contribución de dichas fuerzas sobre cada uno de ellos.

El resultado de esta aplicación consiste en una serie de archivos cuyos nombres comienzan con la cadena "*deformable-object*", que contienen las posiciones finales de las partículas de la malla.

A continuación, se expone que mecanismos se han utilizado para paralelizar esta aplicación en cada uno

de los modelos de programación lineal de los que constaba el *benchmark* P^3ARSEC:

- **Pthreads**: Se utiliza un *pool* de hilos para ejecutar las tareas en paralelo. Este modelo implementa varios archivos auxiliares que posibilitan el uso de ese *pool* que son: *THREAD_ARRAY_LOCK.cpp, THREAD_ARRAY_LOCK.h, THREAD_CONDITION.h, THREAD_DIVISION_PARAMETERS.cpp, THREAD_DIVISION_PARAMETERS.h, THREAD_LOCK.h, THREAD_POOL.cpp, THREAD_POOL.h, THREAD_POOL_ALAMERE.cpp y THREAD_POOL_SINGLE.spp*, todos ellos contenidos en la carpeta Public_Library/Thread_Utilities.

- **FastFlow**: Se implementan varios patrones de tipo *Parallel_for*, con lo que se consigue la explotación del paralelismo de datos.

Implementación con GrPPI

La implementación usando *GrPPI* es muy similar a la de *FastFlow* ya que se usa el patrón equivalente a su *Parallel_for*. Esta aplicación consta de una gran cantidad de funciones que deben ser paralelizadas por lo que la cantidad de patrones usados asciende a 21, por esta razón a continuación solo se muestra uno de ejemplo.

```
e.set_concurrency_degree(pool.number_of_threads);
grppi::parallel_for(e, 1, pool.number_of_threads + 1, 1, chunksize,
       [&](int i){
              ZERO_OUT_ENSLAVED_POSITION_NODES_HELPER<T> helper;
              helper.X = &X;
              helper.partition_attached_nodes = & ( (*attached_nodes_parallel) (i));
              Zero_Out_Enslaved_Position_Nodes_Helper(0, (void*) &helper);
              // Thread id = 0, not used by the function.
       }
);
```

Listado 4.17: *Código de implementación de uno de los patrones parallel_for de la aplicación Facesim*

4.2.5. Ferret

Ferret es una aplicación desarrollada por la Universidad de Princeton para el ámbito de la búsqueda de similitudes usando datos ricos en características como audio, vídeo, imágenes, etc. Este programa se divide en seis etapas: entrada, segmentación de imágenes de consulta, extracción de características, indexación

de los conjuntos candidatos, clasificación y salida. Cada una de las etapas intermedias (sin contar la de entrada y salida) consta de su propio conjunto de threads, como se puede ver en la figura inferior.

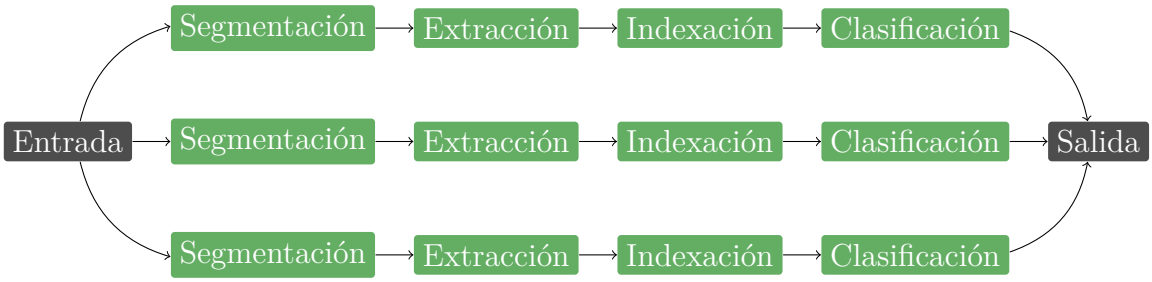


Figura 4-2: *Diagrama de etapas de la aplicación Ferret.*

- La etapa de *segmentación de imágenes de consulta* consiste en la descomposición de cada una de las imágenes de la base de datos en áreas separadas que no se superponen. Estas áreas por lo general suelen representar objetos diferentes de la imagen. Este paso permite dar menor importancia a aquellas partes de la imagen que no son relevantes para el análisis.

- La etapa de *extracción de características* como su propio nombre indica es la encargada de extraer un vector de características de 14 dimensiones para cada uno de los segmentos generados en la etapa anterior. Este vector de características en realidad es una descripción matemática multidimensional en la que se codifican cada una de las propiedades de la imagen, tales como el color, la forma del objeto, el área, etc.

- La etapa de *indexación de los conjuntos candidatos* utiliza los vectores de características generados en la etapa anterior para acceder a la base de datos de imágenes (organizada como un conjunto de tablas hash indexadas con LSH (locality-sensitive hash)) y obtener los conjuntos candidatos. Estos conjuntos no pueden contener una cantidad de imágenes mayor al doble de la cantidad que puede aparecer en la clasificación final.

- La etapa de *clasificación* en la que se calcula una estimación de similitud y ordena las imágenes según el valor calculado. Dicha estimación se calcula mediante la ponderación de las distancias entre los segmentos de la imagen consulta y las imágenes candidatas. De esta forma se obtiene un ranking de las imágenes ordenadas en función de su similitud con la imagen de consulta y finalmente se pueden devolver tantas como se indique en la entrada al programa.

A continuación, se muestra una tabla en la que se exponen los modelos de programación paralela para los cuales la aplicación Ferret posee implementación:

Programa	Modelo de Programación						
	Pthreads	OpenMP	TBB	FastFlow	GrPPI_Native	GrPPI_OpenMP	GrPPI_TBB
Ferret	✓	✗	✓	✓	✓	✗	✓

Tabla 4.5: *Modelos de programación paralela para los que tiene implementación la aplicación Ferret*

Esta aplicación consta de varios archivos de bibliotecas comunes a todas las implementaciones y uno específico para cada modelo, en el caso de la implementación secuencial el archivo se denomina *ferret-serial.c*. *FastFlow y GrPPI* tienen cuatro versiones del código de forma que tienen cuatro archivos diferentes denominados *ferret-ff-farm.cpp, ferret-ff-farm-optimized.cpp, ferret-ff-farmofpipes.cpp y ferret-ff-pipeoffarms.cpp* en el caso de *FastFlow* y *ferret-grppi-farm.cpp, ferret-grppi-farm-optimized.cpp, ferret-grppi-farmofpipes.cpp y ferret-grppi-pipeoffarms.cpp* en el caso de *GrPPI*.

Implementación

Cada uno de los archivos ya mencionados tienen su propio *main* al cual le llegan las siguientes entradas: el directorio para la base de datos de imágenes, la dirección de las consultas, el número de imágenes que se devuelven al final, la profundidad máxima y el archivo de salida. Este *main* en la versión secuencial llama al método *scan* el cual navega por los distintos directorios con ayuda de las funciones *scan_dir y dir_helper*.

Cada vez que se detecta un archivo, se llama a la función *do_query* que es la que lleva a cabo el resto de las etapas (segmentación de imágenes de consulta, extracción de características, indexación de los conjuntos candidatos, clasificación y salida).

Las partes potencialmente paralelizables son las etapas intermedias, ya que las operaciones realizadas sobre distintas imágenes se pueden realizar simultáneamente sin problema. A continuación, se muestra la forma en la que se ha implementado ese paralelismo en los modelos de los que ya constaba el *benchmark* P^3ARSEC:

- **Pthreads**: Se implementa una cola por cada una de las etapas (incluyendo las de entrada y salida). Se crean tantos hilos como se especifican en la ejecución para cada una de las etapas asignándoles a dichos hilos la etapa a la que van a destinar. En este caso a diferencia de la implementación secuencial hay una función por etapa: *t_load, t_seg, t_extract, t_vec_desc, t_rank_desc y t_out_desc*, manteniendo las funciones secuenciales *scan_dir y dir_helper*.

- **TBB**: Al igual que en el caso anterior hay una función por cada una de las etapas. Para la primera etapa y la última (entrada y salida) utiliza la función *tbb::filter (SERIAL_OUT_OF_ORDER_FILTER)*, mientras que para las etapas intermedias usa la función *tbb::filter (parallel)*.

- **FastFlow**: En este caso, como ya se ha comentado hay cuatro versiones del código:

- *Farm*: en esta versión crea una clase *CollapsedPipeline* en la que llama a todas las clases asociadas con las etapas intermedias en orden. Utiliza dicha clase como los trabajadores de su patrón *Farm* y establece la etapa de entrada como el emisor y la de salida como el recolector.
- *Farm Optimizado*: esta versión es muy similar a la anterior, salvo porque las etapas intermedias en lugar de estar implementadas como clases son estructuras.
- *Pipeline de Farms*: en esta versión se establece un patrón *Farm* por cada una de las etapas intermedias y luego un patrón *Pipeline* al que se le insertan como etapas tanto la inicial y la final como los *Farms* creados anteriormente.
- *Farm de Pipelines*: en esta última versión se crea un patrón *Pipeline* en el que se insertan cada una de las llamadas de las funciones asociadas a las etapas y luego se crea un patrón *Farm* en el que se introducen la etapa de entrada como emisor, el *Pipeline* anterior como trabajadores y la etapa de salida con recolector.

Implementación con GrPPI

En el caso de *GRPPI*, se han tratado de imitar las cuatro versiones del modelo *FastFlow*. En primer lugar, se va a mostrar la implementación para la versión del *farm*, que es la que se puede ver a continuación. El código relacionado con *GrPPI* para la versión 2 es el mismo que el de la versión 1, por ello éste solo se muestra una vez.

```
grppi::pipeline(e, [&]() mutable -> optional<struct load_data*> {
       if(j<data.size()) {
              struct load_data* aux = data[j];
              j++;
              return aux;
       } else return {};
   }, grppi::farm(nthreads,[&](struct load_data* task) {
        struct seg_data *seg = segment_function(task);
        struct extract_data *extract = extract_function(seg);
        struct vec_query_data *vec = vec_function(extract);
        return rank_function(vec);
    }), [&](struct rank_data* task) {
       out_function((struct rank_data*) task);}
);
```

Listado 4.18: *Código de la versión 1 de la implementación (patrón farm) para la aplicación Ferret.*

La versión 3, es la correspondiente con el patrón *pipeline* como principal y varios patrones de tipo *farm* en su interior:

```
grppi::pipeline(e,
    [&]() mutable -> optional<struct load_data*> {
        if(j<data.size()) {
                struct load_data* aux = data[j];
                j++;
                return aux;
         }
         else return {};
    },
    grppi::farm(nthreads,[&](struct load_data* task) {
                return segment_function(task);
    }),
    grppi::farm(nthreads,[&](struct seg_data* task) {
                return extract_function(task);
    }),
    grppi::farm(nthreads,[&](struct extract_data* task) {
                return vec_function(task);
    }),
    grppi::farm(nthreads,[&](struct vec_query_data* task){
                return rank_function(task);
    }),
    [&](struct rank_data* task) {
        struct rank_data * rank = (struct rank_data*) task;
        out_function (rank);
     }
);
```

Listado 4.19: *Código de la versión 3 de la implementación (patrón pipeline con varios farms en su interior) para la aplicación Ferret.*

Y, la versión 4, que consta de los mismos tipos de patrones que la anterior pero anidados al revés, es decir, en esta versión hay un *Farm* como patrón principal y un patrón de tipo *Pipeline* con diversas etapas en su interior:

```
/* Load Stage */
grppi::pipeline(e,
    [&]() mutable -> optional<struct load_data*> {
                if(j<data.size()) {
                        struct load_data* aux = data[j];
                        j++;
                        return aux;
                }
                else return {};
    },
    grppi::farm(nthreads,
                grppi::pipeline(
                        [](struct load_data* load) { /* Segment Stage */
                                return segment_function(load);},
                         [&](struct seg_data* task) { /* Extract Stage */
                                return extract_function(task);},
                          [&](struct extract_data* task) { /* Vec Stage */
                                return vec_function(task); },
                          [&](struct vec_query_data* task) { /* Rank Stage */
                                return rank_funciton(task);}
                        )
        ),
    [&](struct rank_data* task) { /* Out Stage */
        struct rank_data * rank = (struct rank_data*) task;
        out_function(rank);
    }
);
```

Listado 4.20: *Código de la versión 4 de la implementación (patrón farm con un pipeline en su interior) para la aplicación Ferret.*

4.2.6. Fluidanimate

Fluidanimate es una aplicación del benchmark Intel RMS para el ámbito de la animación, que utiliza una extensión del método hidrodinámico de partículas suavizadas (SPH) para simular un fluido con el fin de realizar una animación interactiva. Para ello se hace uso de las ecuaciones de NavierStokes, que son un

conjunto de ecuaciones que describen el movimiento de los fluidos Newtonianos. Este tipo de fluidos se caracterizan por tener una viscosidad constante y un buen ejemplo de estos es el agua. Este programa deriva la densidad de los campos de fuerza directamente de la ecuación simplificada de Navier-Stokes para la conservación del momento lineal, que es la siguiente:

$$\rho(\frac{\partial v}{\partial t} + v \cdot \nabla v) = -\nabla p + \rho g + \mu \nabla^2 v \tag{4.6}$$

donde la v es la velocidad del campo, ρ es la densidad del campo, p es la presión del campo, g es una fuerza externa y μ es la viscosidad del fluido.

El modelo SPH utiliza partículas para modelar el estado del fluido, estas se colocan en ubicaciones discretas que se interpolan para obtener los valores intermedios mediante el uso de *kernels* de suavizado radial simétrico.

La escena que utiliza esta aplicación es una caja en la que se encuentran las partículas que representan el fluido, como se puede ver en la siguiente imagen:

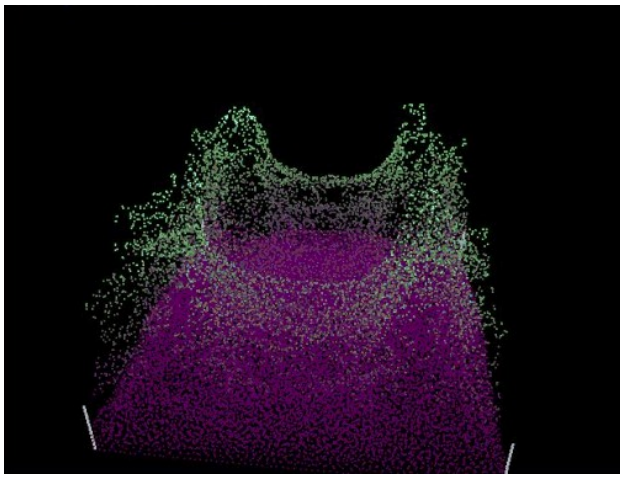


Figura 4-3: *Imagen de las partículas Newtonianas extraída de la página web [1]*

A continuación, se muestra una tabla en la que se exponen los modelos de programación paralela para los cuales la aplicación Fluidanimate posee implementación:

Programa	Modelo de Programación						
	Pthreads	OpenMP	TBB	FastFlow	GrPPI_Native	GrPPI_OpenMP	GrPPI_TBB
Fluidanimate	✓	✗	✗	✓	✓	✓	✓

Tabla 4.6: *Modelos de programación paralela para los que tiene implementación la aplicación Fluidanimate*

Implementación

Esta aplicación tiene como entrada un conjunto de partículas Newtonianas que interactúan entre sí, así como el número de *frames* que se van a emplear.

Facesim se compone de cinco fases generales, algunas de ellas divididas en subfases. Primero tiene lugar la reconstrucción del índice espacial, las partículas solo pueden interactuar hasta una distancia máxima lo que permite limitar el número de partículas que se deben analizar gracias al uso de una estructura de indexación espacial. Esta primera fase se divide en dos subfases que se implementan en las funciones *ClearParticlesMT*, encargada de restablecer todos los valores de las partículas a 0 y *RebuildGridMT* que crea la estructura mencionada.

La segunda fase consiste en estimar el valor de las densidades de las partículas teniendo en cuenta la proximidad de sus partículas vecinas, cuanto más próximas estén las partículas, mayor será la densidad. Esta segunda fase se divide en tres subfases que se implementan en las funciones *InitDensitiesandForcesMT* encargada de inicializar las densidades a 0.0, *ComputeDensitiesMT* y *ComputeDensities2MT* destinadas al cálculo del valor de dichas densidades.

La tercera fase consiste en evaluar la presión, la gravedad, la viscosidad y las densidades ya calculadas para determinar el valor de la fuerza, para ello se usa la función *ComputeForcesMT*.

La cuarta fase se encarga de actualizar las fuerzas calculadas para manejar las colisiones de las partículas y la caja en la que se encuentran, para ello se usa la función *ProcessCollisionsMT*.

Y, por último, la quinta fase se actualizan las posiciones de las partículas usando las fuerzas anteriores para calcular la velocidad de estas. Esto se realiza mediante un integrador Verlet implementado en la función *AdvancedParticlesMT*.

A continuación, se muestra como se ha implementado el paralelismo para esta aplicación en cada uno de los modelos paralelos de los que ya constaba el *benchmark*:

- **Pthreads**: Se crean tantos hilos como se especifique en la entrada, asociando a cada uno de ellos el ciclo de fases completo.

- **TBB**: Se establecen cada una de las funciones mencionadas como tareas de *TBB* (*tbb::task*). Además de esa función de *TBB* también se usan *tbb::task::spawn_root_and_wait* y *tbb::task::allocate_root*.

- **FastFlow**: Se utiliza varias veces el patrón *Parallel_for*.

Implementación con GrPPI

En el caso de *GrPPI*, se usan nueve patrones del tipo *Parallel_for* de forma que solo se va a mostrar un ejemplo de estos a continuación.

```
grppi::parallel_for(e2, 0, NUM_GRIDS, 1, CHUNKSIZE,
    [&](auto tid){
    for(int iz = grids[tid].sz; iz < grids[tid].ez; ++iz)
      for(int iy = grids[tid].sy; iy < grids[tid].ey; ++iy)
        for(int ix = grids[tid].sx; ix < grids[tid].ex; ++ix){
          int index = (iz*ny + iy)*nx + ix;
          Cell *cell = &cells[index];
          int np = cnumPars[index];
          for(int j = 0; j < np; ++j){
            cell->density[j % PARTICLES_PER_CELL] = 0.0;
            cell->a[j % PARTICLES_PER_CELL] = externalAcceleration;
            //move pointer to next cell in list if end of array is reached
            if(j % PARTICLES_PER_CELL == PARTICLES_PER_CELL-1) cell = cell->next;
          }
        }
    }
);
```

Listado 4.21: *Código de implementación del patrón parallel_for para la aplicación Raytrace.*

4.2.7. Raytrace

Raytrace es una aplicación del benchmark de Intel RMS para el ámbito del renderizado que utiliza una variación del método de trazado de rayados para generar imágenes realistas en 3D. El método del trazado de rayos genera la escena que se muestra mediante sombras, reflexiones y refracciones de los rayos de luz. Pero esta técnica en lugar de trazar todos los rayos que salen desde las fuentes de luz traza solo aquellos que llegan a los ojos del observador, como se puede ver en la Figura 4-4, con lo que se consigue disminuir el computo necesario.

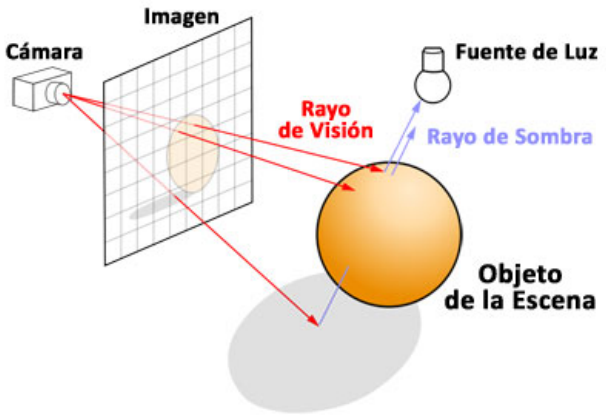


Figura 4-4: *Principio básico del método de trazado de rayos Raytrace, extraída de la página web [2]*

Esta técnica crea tres tipos de rayos: rayos de reflexión, que son aquellos que se crean cuando un rayo de luz incide sobre una superficie brillante (reflexión especular) o mate (reflexión difusa), rayos de refracción, que son los que se crean cuando la luz incide sobre una superficie transparente y la atraviesa y los rayos de sombra, que se usan para determinar las zonas visibles de la imagen.

Como ya se ha comentado, esta aplicación no emplea el método tradicional sino una variación de este que se emplea generalmente en los gráficos en tiempo real, en la cual se sacrifica el realismo de la imagen por la velocidad.

Esta variación consta de un árbol BVH (Bounding Volume Hierarchy) en el que cada uno de los nodos hoja representa un objeto de la escena contenido en un volumen delimitador. Los nodos intermedios son volúmenes delimitadores de mayor tamaño que contienen a todos sus nodos hojas y el nodo raíz es el que contiene la escena completa. Este árbol sirve para determinar los puntos de intersección con los rayos de forma más rápida, ya que permite descartar algunas partes de la escena.

A continuación, se muestra una tabla en la que se exponen los distintos modelos de programación paralela para los cuales la aplicación Raytrace posee implementación:

Programa	Modelo de Programación						
	Pthreads	OpenMP	TBB	FastFlow	GrPPI_Native	GrPPI_OpenMP	GrPPI_TBB
Raytrace	✓	✗	✗	✓	✓	✓	✓

Tabla 4.7: *Modelos de programación paralela para los que tiene implementación la aplicación Raytrace*

Esta aplicación tiene varios archivos de código, pero solo es necesario cambiar uno de ellos para pasar de la versión secuencial a la paralela, ese archivo es *render.cxx*. En este archivo se encuentran todas las implementaciones paralelas separadas por macros del tipo *#ENABLE_GRPPI*, de forma que si esta macro está activada significa que se quiere ejecutar usando *GrPPI*.

Implementación

En este caso la entrada a la aplicación consiste en un objeto complejo formado por triángulos, el número de polígonos que se van a usar, el número de *frames* y la resolución para la salida.

El programa analiza las entradas y llama tantas veces al método *render* como el número de *frames* que se le introduce con entrada. Dicho método es el que llama al método de inicio de la clase *render.cxx* (*lrtRenderFrame*), el cual ejecuta la función *renderFrame* que crea los hilos en el caso de que se necesiten y llama al método *renderTile* con el modo de la geometría, el plan de rayos y la resolución como argumentos. Este método itera por cada uno de los píxeles (el número de píxeles dependerá de la resolución especificada en la entrada) y traza los rayos que pasan por ellos para determinar la cantidad de luz o sombra que tendrán.

Esta aplicación consta de dos modos de geometría: *MINIRT_POLYGONAL_GEOMETRY* y *MINIRT_SUBDIVISION_SURFACE_GEOMETRY* y dependiendo del modo establecido se usan distintos planes de rayos: *RAY_PACKET_LAYOUT_TRIANGLE* o *RAY_PACKET_LAYOUT_SUBDIVISION*. Como los objetos de entrada están formados por triángulos se suele usar el modo poligonal y el plan basado en triángulos.

Como se ha comentado, el método *renderFrame* es el encargado de crear los hilos, de esta forma en lugar de llamar al método *renderTile* con la resolución completa, se divide en tantas partes como el número de hilos que se especifique en la entrada y se llama a dicho método con la fracción de la resolución asignada a cada uno de los hilos. A continuación, se expone la manera en la que se ha implementado dicha paralelización con cada uno de los modelos de programación de los que ya constaba el *benchmark* P^3ARSEC:

- **Pthreads**: Simplemente se crean los hilos usando el método *createThreads*, luego se inicializan con el método *startThreads* y por último se sincronizan usando el método *waitForAllThreads*.
- **FastFlow**: Se usa el patrón *Parallel_for* para hacer la división de la resolución y asignar a cada uno de los hilos la fracción adecuada, teniendo en cuenta el resto en el caso de que la división no sea exacta.

Implementación con GrPPI

En el caso de *GrPPI* se utiliza el patrón equivalente al de FastFlow (*Parallel_for*), cuya implementación se puede ver a continuación:

```
grppi::parallel_for(e, 0, m_threadData.maxTiles, 1, 58, [&](int index) {
       int sx = (index \% tilesPerRow)*TILE_WIDTH;
       int sy = (index / tilesPerRow)*TILE_WIDTH;
```

```
        int ex = min(sx+TILE_WIDTH,m_threadData.resX);
        int ey = min(sy+TILE_WIDTH,m_threadData.resY);
        if (m_geometryMode == MINIRT_POLYGONAL_GEOMETRY)
            this -> renderTile<StandardTriangleMesh,RAY_PACKET_LAYOUT_TRIANGLE>(
                m_threadData.frameBuffer,sx,sy,ex,ey);
        else if (m_geometryMode == MINIRT_SUBDIVISION_SURFACE_GEOMETRY)
            this -> renderTile<DirectedEdgeMesh,RAY_PACKET_LAYOUT_SUBDIVISION>(
                m_threadData.frameBuffer,sx,sy,ex,ey);
        else
            FATAL("unknown mesh type");
        return index;
    }
);
```

Listado 4.22: *Código de implementación del patrón parallel_for para la aplicación Raytrace.*

4.2.8. Streamcluster

Streamcluster es un kernel RMS desarrollado por la Universidad de Princeton para el ámbito de la minería de datos que trata de resolver el problema de agrupación o *clustering* en línea.

Dado un flujo de puntos de entrada, determina cual es el número de medianas adecuado de tal forma que cada vez que llegue un nuevo punto se asigne al centro más cercano posible. Dicha agrupación se evalúa mediante el uso de una métrica, la suma de las distancias cuadradas.

Es posible crear un nuevo centro para la agrupación en cualquier momento por ello este *kernel* emplea gran parte de su tiempo de ejecución en evaluar la ganancia que tendría crear dicho centro. Para ello se coge la solución actual y evalúa el ahorro que supondría coger el nuevo punto que llega y transformarlo en un centro de forma que se puedan reasignar varios de los puntos anteriores, mejorando así la agrupación.

A continuación, se muestra una tabla en la que se exponen los modelos de programación paralela para los cuales el *kernel* Streamcluster posee implementación:

Programa	Modelo de Programación						
	Pthreads	OpenMP	TBB	FastFlow	GrPPI_Native	GrPPI_OpenMP	GrPPI_TBB
Streamcluster	✓	✗	✗	✓	✓	✓	✓

Tabla 4.8: *Modelos de programación paralela para los que tiene implementación la aplicación Streamcluster*

En el caso de *Streamcluster* solo es necesario un archivo de código *streamcluster.cpp* común para todos los

modelos implementados. Aunque hay modelos (*Pthreads y FastFlow*) que requieren el uso de dos archivos más *parsec_barrier.hpp y parsec_barrier.cpp*.

Implementación

Este *kernel* tiene como entrada el número máximo y mínimo de centros que se pueden usar, la dimensión de un punto de datos, el número de puntos de datos, el número de puntos que se pueden manejar por unidad de tiempo y los archivos de entrada y salida.

Una vez se han comprobado las entradas se llama al método principal *StreamCluster* que es el que lleva a cabo el papel de *main*, éste llama a la función *localSearch* va ejecutando a las distintas funciones que componen el *kernel*.

Este programa está formado por varias funciones que se reimplementan dependiendo del modelo de programación que se esté usando, éstas son: *localSearch, pkmedian, pspeedy, pFL y pgain*.

- La función *pgain* es la encargada de calcular la ganancia de añadir un nuevo centro.
- La función *pkmedian* calcula un valor aproximado para los puntos de la *k-mediana*.
- La función *pspeedy* hace una ejecución rápida sobre los puntos para devolver el coste total de la solución.
- La función *pFL* calcula la posición de instalación mediante una búsqueda local.

A continuación, se muestra como se ha implementado el paralelismo con cada uno de los modelos del *benchmark*:

- **Pthreads**: Se han creado tantos hilos como se especificaba en la entrada mediante la función *pthread_create* y para sincronizar los hilos se han usado barreras (*pthread_barrier_t*).
- **TBB**: Se han usado las funciones de *TBB*: *tbb::task::spawn_root_and_wait, tbb::task::allocate_root, tbb::blocked_range, tbb::parallel_reduce, tbb::parallel_for, tbb::task_list y tbb::task*
- **FastFlow**: Se han usado patrones de tipo *Parallel_for* y un patrón de tipo *Parallel_for_reduce*.

Implementación con GrPPI

En el caso de *GrPPI*, al igual que con *FastFlow* se ha implementado el paralelismo haciendo uso de siete patrones de tipo *Parallel_for* y un patrón de tipo *reduce*. A continuación, se muestra el código paralelo de la función *pkmedian* que es la que contiene el patrón *reduce*.

```
grppi::parallel_for(e, 0, (int) points->num, 1,
  [&](auto idx) {
     hizs[idx] += dist(points->p[idx], points->p[0],ptDimension)*points->p[idx].weight;
  }
);
double sum = 0.0, v = 0.0;
grppi::reduce(e, begin(vec1), end(vec1), 0,
  [&](auto i, auto j) {
     sum += hizs[i];
     sum += hizs[j];
     return 0;
  }
);
```

Listado 4.23: *Código de implementación del patrón parallel_for para la aplicación Raytrace.*

Capítulo 5

Evaluación del Rendimiento

En este capítulo se exponen las características principales de la arquitectura que se ha usado para la evaluación del rendimiento de las distintas aplicaciones (Sección 5.1), así como la metodología que se ha seguido para sacar cada uno de los tiempos que se van a mostrar (Sección 5.2). Además, se exponen los resultados obtenidos para cada una de las aplicaciones ya explicadas (Sección 4.2) y las conclusiones que se pueden extraer de dichos resultados (Sección 5.3).

5.1. Descripción del Entorno

La evaluación se ha llevado a cabo en una plataforma de servidor con una arquitectura NUMA compuesta por 2x Intel Xeon Ivy Bridge E5-2695 v2 con un total de 24 núcleos de 2.4GHz. Además, esta plataforma consta de una cache L3 de 30 MB y 128 GB de RAM DDR3. En cuanto al sistema operativo, dicha plataforma opera sobre Linux Ubuntu 14.04.2 LTS con la versión 3.13.0-57 del kernel y la versión 6.3.0 del compilador.

En concreto la arquitectura NUMA utilizada consta de dos nodos, 0 y 1. En cada uno de esos nodos se agrupan 12 de los 24 núcleos de los que se compone la plataforma utilizada, es decir, que los núcleos del 0 al 11 están asociados al nodo NUMA 0 y los núcleos del 12 al 23 en el nodo 1. Como ya se ha comentado el tiempo de acceso a la memoria no local, es mayor que el tiempo de acceso a la memoria local, pero el tiempo concreto depende de la máquina utilizada.

nodo	0	1
0	10	21
1	21	10

Tabla 5.1: *Matriz de distancias relativas entre los nodos NUMA.*

Como se puede ver en la tabla anterior el tiempo que tarda cualquiera de los nodos en acceder a la memoria del nodo contrario es más del doble de lo que tarda en acceder a su propia memoria local, por ello

es importante para el rendimiento que los nodos accedan el menor número de veces posible a la memoria no local.

Para atenuar el incremento de tiempo de ejecución que se produce cuando el proceso accede a la memoria no local se han usado los siguientes dos comandos:

- *numactl:* este comando permite alterar la política de procesos y memoria compartida de la arquitectura NUMA. Este comando ejecuta con una política de asignación de memoria o NUMA especifica. La política establecida mediante el comando la heredan todos sus hijos. Este comando también puede establecer una política persistente para archivos o segmentos de memoria compartida. El comando numactl tiene una gran variedad de opciones de las cuales se han probado -i, -m, -N, -l y –preferred, para intentar determinar cuál sería la opción que atenúa en mayor media las características de la arquitectura NUMA y finalmente la opción seleccionada es -i.

  La opción -i (–interleave=nodes) permite establecer una política de intercalación de memoria, es decir, que la memoria se asignará mediante un round robin de los nodos disponibles. En concreto el comando probado es:

  $$numactl \quad -i \quad 0-1$$

- *taskset:* este comando permite modificar la afinidad entre proceso y CPU. La afinidad es una propiedad del planificador que permite enlazar un proceso con un conjunto de CPUs del sistema, de forma que dicho proceso solo se ejecutará en dicho conjunto de CPUs. Este comando tiene varias opciones (-p, -c, -h y -v), sin embargo, la que se ha usado en este caso es -c (–cpu-list), mediante la cual se puede especificar dicha lista de CPUs separada por comas o mediante rangos. En concreto el comando utilizado es:

  $$taskset \quad -c \quad 0-nthreads$$

  siendo nthreads, el número de hilos que se usan en dicha ejecución del programa.

5.2. Metodología

Para realizar la evaluación del rendimiento de las aplicaciones explicadas en la sección 4.2 se han colocado una serie de *timers* de la clase *chrono* de C++ para medir el tiempo de ejecución de cada una de las aplicaciones. El objetivo principal es poder hacer una comparación del rendimiento obtenido para cada una de las bibliotecas de paralelismo empleadas en dichas aplicaciones, para lo cual se han desarrollado scripts de ejecución en BASH.

Estos scripts de BASH lanzaban para cada una de las aplicaciones 5 ejecuciones para cada número de hilos del 1 al 24 (que es el número de núcleos de los que dispone la plataforma) y biblioteca paralela, es decir, que si la aplicación Raytrace ha sido implementada con las bibliotecas de paralelismo pthreads, fastflow, grppi-openmp, grppi-tbb y grppi-native, entonces el script lanzaría 5 ejecuciones para la biblioteca pthreads con 1 hilo, luego 5 ejecuciones con 2 hilos y así hasta que llegara a 24 hilos, y luego cambiaría a la siguiente biblioteca (fastflow en este caso). Así se evita tener que lanzar cada una de esas ejecuciones de forma manual. Se ejecuta cada una de las configuraciones 5 veces para poder obtener una media del tiempo de ejecución que sea más precisa. Además de la media también se calcula la varianza y la desviación estándar, de forma que se pueda analizar qué tan dispersos se encuentran los datos alrededor de esa media.

5.3. Resultados de la Evaluación

En esta sección se exponen los resultados obtenidos de la evaluación del rendimiento de la biblioteca de patrones paralelos *GrPPI* para algunas de las aplicaciones del *benchmark* de P^3ARSEC, en concreto las explicadas en la sección 4.2.

En primer lugar, cabe decir que durante el periodo de obtención de los tiempos se tuvieron que hacer algunos cambios en la implementación interna de la versión de *GrPPI* que usa hilos de C++ (*GrPPI_Native*), ya que había aplicaciones dentro de las seleccionadas en las que la distribución de los datos no era homogénea, esto suponía que algunos hilos tuvieran más computo que otros y aumentaran el tiempo de ejecución general. La modificación que se hizo fue adaptar dicha versión de *GrPPI* para que siguiera una planificación estática en la que se pudiera especificar en para cada una de las aplicaciones que empleaban el patrón *Parallel_for* el tamaño más adecuado para hacer las divisiones de datos (*chunksize*). El análisis realizado para determinar dicho tamaño se explica antes de empezar con la evaluación de cada una de las aplicaciones.

Por otro lado, en cuanto al conjunto de datos seleccionado para realizar la evaluación, decir que entre los definidos en la Sección 4.2 se ha usado el conjunto *native*, puesto que los propios creadores del *benchmark* indican que es el más adecuado para realizar una evaluación de rendimiento. El contenido de esta entrada va a variar dependiendo de la aplicación estudiada por ello en cada una de las subsecciones siguientes se explica de que argumentos consta el conjunto seleccionados en ese caso concreto.

5.3.1. Blackscholes

En el caso de la aplicación *Blackscholes*, dependiendo del conjunto de entrada que se escoja el número de opciones que se deben calcular varía. Al escoger el conjunto *native* el número de opciones a calcular es de 10.000.000.

A continuación, se expone la evaluación de las dos versiones mencionadas en la Sección 4.2.1: la versión 1 formada por un único patrón de paralelismo, el *Parallel_for* y la versión 2 formada por dos patrones anidados, un *Farm* (que conlleva el uso del patrón *Pipeline*) y un *Map*.

Versión 1: Parallel_for

En esta versión al utilizar el patrón *Parallel_for*, fue necesario realizar un análisis del valor del *chunksize*, para ello se realizaron diversas ejecuciones con 6, 12, 18 y 24 hilos, variando en cada caso el valor del *chunksize* en 100 unidades en el rango de 1 a 2000. Los resultados se pueden ver en la siguiente gráfica:

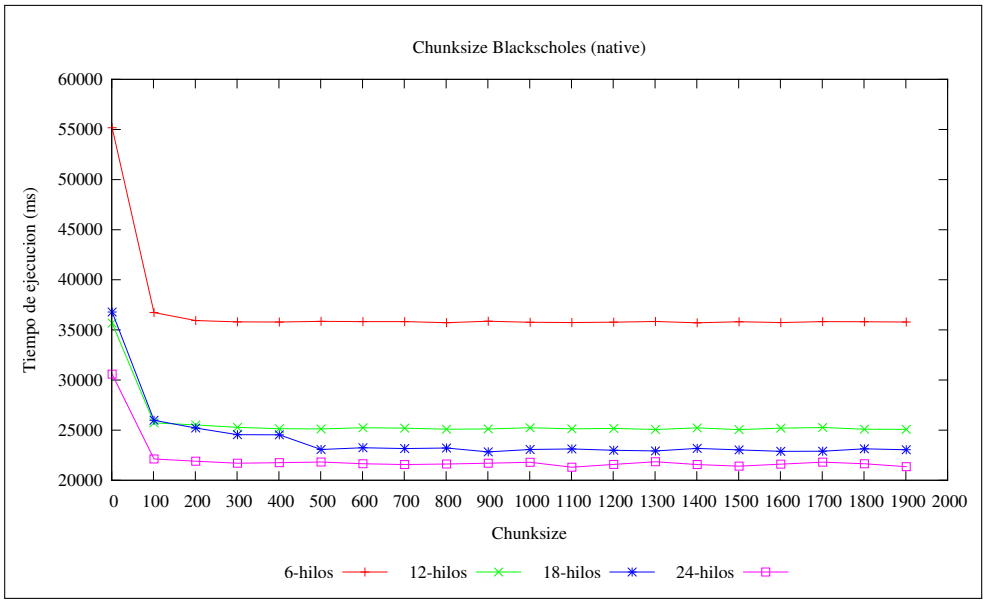


Figura 5-1: *Gráfica de tiempos de ejecución para los valores del chunksize usando la implementación 1 de GrPPI Native para la aplicación Blackscholes.*

Como se puede observar en la gráfica anterior, a partir del valor 500 el tiempo de ejecución se estabiliza, de forma que se puede pasar a analizar simplemente el rango entre 1 y 500. En éste la línea correspondiente con las ejecuciones usando 18 hilos tiene su mínimo en el valor 500, por lo que se decidió usar dicho valor para el resto de las ejecuciones de esta versión del programa.

Los resultados de las medidas de dispersión aplicadas a los tiempos de ejecución obtenidos con esta versión del programa se muestran en la siguiente tabla:

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
1	serial	104308	238,7865155	57019	$\mu \in$ [104132,3484 ; 104483,6516]
2	fastflow	63778	90,29673305	8153,5	$\mu \in$ [63711,57764 ; 63844,42236]
2	pthreads	57152	27,72183255	768,5	$\mu \in$ [57131,60779 ; 57172,39221]
2	openmp	57281	192,0898227	36898,5	$\mu \in$ [57139,69855 ; 57422,30145]
2	tbb	57183,8	38,84198759	1508,7	$\mu \in$ [57155,2278 ; 57212,3722]
4	fastflow	37069,6	162,3539344	26358,8	$\mu \in$ [36950,1723 ; 37189,0277]
4	pthreads	33313,2	43,14162723	1861,2	$\mu \in$ [33281,46498 ; 33344,93502]
4	*grppi_native*	45707,4	143,740391	20661,3	$\mu \in$ [45601,66444 ; 45813,13556]
4	*grppi_openmp*	45919,2	609,3485866	371305,7	$\mu \in$ [45470,96263 ; 46367,43737]
4	*grppi_tbb*	45241,8	110,4658318	12202,7	$\mu \in$ [45160,54123 ; 45323,05877]

4	openmp	33339,8	80,41579447	6466,7	$\mu \in$ [33280,64607 ; 33398,95393]
4	tbb	33372,8	50,02699271	2502,7	$\mu \in$ [33336,0001 ; 33409,5999]
6	fastflow	27335,6	167,1834322	27950,3	$\mu \in$ [27212,61972 ; 27458,58028]
6	pthreads	24673,2	44,1837074	1952,2	$\mu \in$ [24640,69842 ; 24705,70158]
6	*grppi_native*	35792,6	149,9326515	22479,8	$\mu \in$ [35682,30941 ; 35902,89059]
6	*grppi_openmp*	35370,6	552,9767626	305783,3	$\mu \in$ [34963,82979 ; 35777,37021]
6	*grppi_tbb*	35075,2	117,4210373	13787,7	$\mu \in$ [34988,82498 ; 35161,57502]
6	openmp	24745,8	127,6232737	16287,7	$\mu \in$ [24651,9202 ; 24839,6798]
6	tbb	26848,2	520,7894968	271221,7	$\mu \in$ [26465,10678 ; 27231,29322]
8	fastflow	22253,6	403,4975836	162810,3	$\mu \in$ [21956,78681 ; 22550,41319]
8	pthreads	20079,2	246,6286682	60825,7	$\mu \in$ [19897,77973 ; 20260,62027]
8	*grppi_native*	30319	79,35048834	6296,5	$\mu \in$ [30260,62971 ; 30377,37029]
8	*grppi_openmp*	29099	433,2003001	187662,5	$\mu \in$ [28780,33747 ; 29417,66253]
8	*grppi_tbb*	29521	102,4597482	10498	$\mu \in$ [29445,63052 ; 29596,36948]
8	openmp	19928,8	6,418722614	41,2	$\mu \in$ [19924,07838 ; 19933,52162]
8	tbb	20304,4	425,733837	181249,3	$\mu \in$ [19991,2298 ; 20617,5702]
10	fastflow	18984,8	154,1418178	23759,7	$\mu \in$ [18871,41314 ; 19098,18686]
10	pthreads	17154,6	38,31840289	1468,3	$\mu \in$ [17126,41295 ; 17182,78705]
10	*grppi_native*	27528,6	124,2569918	15439,8	$\mu \in$ [27437,19644 ; 27620,00356]
10	*grppi_openmp*	25955,4	485,1508013	235371,3	$\mu \in$ [25598,52263 ; 26312,27737]
10	*grppi_tbb*	26189,4	127,6217066	16287,3	$\mu \in$ [26095,52136 ; 26283,27864]
10	openmp	17150,4	25,23489647	636,8	$\mu \in$ [17131,83719 ; 17168,96281]
10	tbb	18192,2	605,0485105	366083,7	$\mu \in$ [17747,12577 ; 18637,27423]
12	fastflow	17116,4	245,8308361	60432,8	$\mu \in$ [16935,56661 ; 17297,23339]
12	pthreads	15289	22,57210668	509,5	$\mu \in$ [15272,39594 ; 15305,60406]
12	*grppi_native*	25191,2	144,3665474	20841,7	$\mu \in$ [25085,00384 ; 25297,39616]
12	*grppi_openmp*	23792	478,3377468	228807	$\mu \in$ [23440,13432 ; 24143,86568]
12	*grppi_tbb*	24114	128,3043257	16462	$\mu \in$ [24019,61922 ; 24208,38078]
12	openmp	15290,2	36,80624947	1354,7	$\mu \in$ [15263,12529 ; 15317,27471]
12	tbb	17063,2	560,3000982	313936,2	$\mu \in$ [16651,04274 ; 17475,35726]
14	fastflow	15258	160,7435846	25838,5	$\mu \in$ [15139,75687 ; 15376,24313]
14	pthreads	19905,6	2082,561116	4337060,8	$\mu \in$ [18373,66616 ; 21437,53384]
14	*grppi_native*	25509,2	212,7268201	45252,7	$\mu \in$ [25352,71796 ; 25665,68204]
14	*grppi_openmp*	22830,2	608,9899014	370868,7	$\mu \in$ [22382,22648 ; 23278,17352]
14	*grppi_tbb*	22594,2	147,3319382	21706,7	$\mu \in$ [22485,82249 ; 22702,57751]
14	openmp	16357,2	1284,654311	1650336,7	$\mu \in$ [15412,20713 ; 17302,19287]
14	tbb	14097,8	120,8519756	14605,2	$\mu \in$ [14008,90118 ; 14186,69882]

16	fastflow	14390,4	193,1561544	37309,3	$\mu \in$ [14248,31416 ; 14532,48584]
16	pthreads	15672,8	1666,963467	2778767,2	$\mu \in$ [14446,58017 ; 16899,01983]
16	*grppi_native*	23859,2	212,1584314	45011,2	$\mu \in$ [23703,13607 ; 24015,26393]
16	*grppi_openmp*	21542,8	357,7935159	128016,2	$\mu \in$ [21279,60677 ; 21805,99323]
16	*grppi_tbb*	21533,2	145,3141425	21116,2	$\mu \in$ [21426,30679 ; 21640,09321]
16	openmp	14364,2	1729,144789	2989941,7	$\mu \in$ [13092,23953 ; 15636,16047]
16	tbb	13259,8	68,5434169	4698,2	$\mu \in$ [13209,3794 ; 13310,2206]
18	fastflow	13387,6	182,3562448	33253,8	$\mu \in$ [13253,45858 ; 13521,74142]
18	pthreads	15900,6	367,0276556	134709,3	$\mu \in$ [15630,61412 ; 16170,58588]
18	*grppi_native*	23039,8	284,8564551	81143,2	$\mu \in$ [22830,25933 ; 23249,34067]
18	*grppi_openmp*	20839,8	523,7873614	274353,2	$\mu \in$ [20454,50154 ; 21225,09846]
18	*grppi_tbb*	20618,6	103,3576316	10682,8	$\mu \in$ [20542,57003 ; 20694,62997]
18	openmp	13451	1599,673717	2558956	$\mu \in$ [12274,27857 ; 14627,72143]
18	tbb	13218	160,8726204	25880	$\mu \in$ [13099,66195 ; 13336,33805]
20	fastflow	12991	367,2716978	134888,5	$\mu \in$ [12720,83461 ; 13261,16539]
20	pthreads	13179,6	1263,513474	1596466,3	$\mu \in$ [12250,15835 ; 14109,04165]
20	*grppi_native*	23036,2	274,0861179	75123,2	$\mu \in$ [22834,582 ; 23237,818]
20	*grppi_openmp*	19775,2	303,1364379	91891,7	$\mu \in$ [19552,21256 ; 19998,18744]
20	*grppi_tbb*	19917,2	69,61106234	4845,7	$\mu \in$ [19865,99404 ; 19968,40596]
20	openmp	13406,8	1548,474475	2397773,2	$\mu \in$ [12267,74078 ; 14545,85922]
20	tbb	12803,4	727,6728661	529507,8	$\mu \in$ [12268,12318 ; 13338,67682]
22	fastflow	12306,2	545,0111925	297037,2	$\mu \in$ [11905,28927 ; 12707,11073]
22	pthreads	12037,6	1286,707348	1655615,8	$\mu \in$ [11091,09691 ; 12984,10309]
22	*grppi_native*	22571,6	149,9759981	22492,8	$\mu \in$ [22461,27752 ; 22681,92248]
22	*grppi_openmp*	19178,6	305,8599353	93550,3	$\mu \in$ [18953,60915 ; 19403,59085]
22	*grppi_tbb*	19310,6	111,8136843	12502,3	$\mu \in$ [19228,34975 ; 19392,85025]
22	openmp	11452	441,5535075	194969,5	$\mu \in$ [11127,19284 ; 11776,80716]
22	tbb	12154	312,3795768	97581	$\mu \in$ [11924,2133 ; 12383,7867]
24	fastflow	11927,8	458,2976107	210036,7	$\mu \in$ [11590,67586 ; 12264,92414]
24	pthreads	10869,8	270,5913894	73219,7	$\mu \in$ [10670,75273 ; 11068,84727]
24	*grppi_native*	21625,2	110,9918916	12319,2	$\mu \in$ [21543,55426 ; 21706,84574]
24	*grppi_openmp*	18238	863,138749	745008,5	$\mu \in$ [17603,07436 ; 18872,92564]
24	*grppi_tbb*	18789,8	58,64895566	3439,7	$\mu \in$ [18746,65778 ; 18832,94222]
24	openmp	11004,8	337,3843506	113828,2	$\mu \in$ [10756,61977 ; 11252,98023]
24	tbb	11964,6	467,5535263	218606,3	$\mu \in$ [11620,6672 ; 12308,5328]

Tabla 5.2: *Medidas de dispersión obtenidas con la implementación 1 de la aplicación Blackscholes.*

Como se puede ver en la tabla anterior ninguna de las versiones de *GrPPI* tiene las medidas de dispersión para el caso de 2 hilos, esto es debido a una restricción establecida a la hora de implementar la aplicación.

A continuación, se muestran esos mismos datos mediante una gráfica para una mejor comparativa de los distintos modelos paralelos:

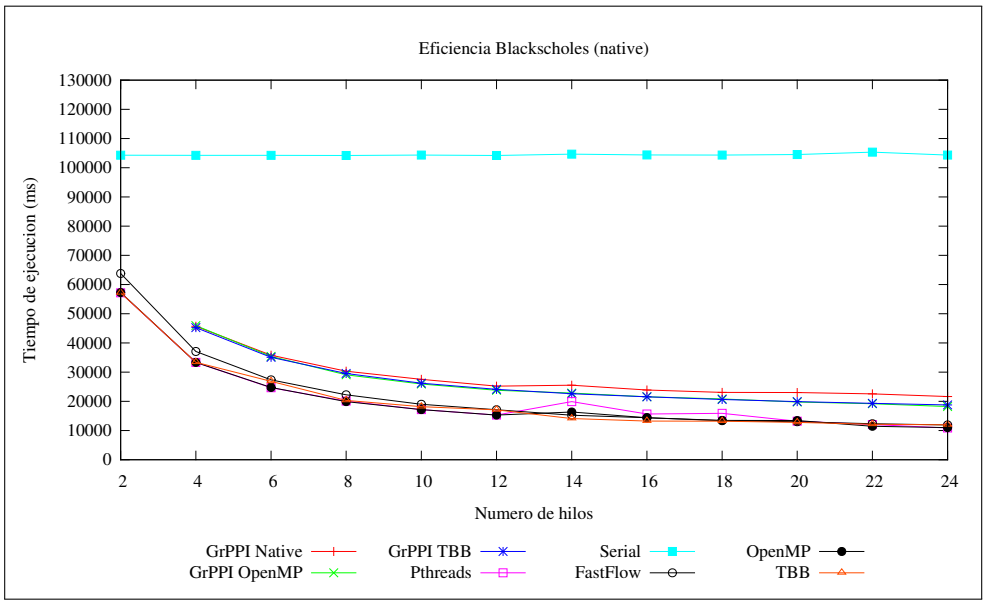


Figura 5-2: *Gráfica de tiempos de ejecución medios para los distintos modelos de programación de la aplicación Blackscholes, usando la implementación de la versión 1 para GrPPI.*

A parte del tiempo de ejecución medio, también se considera el *speedup* o aumento de la eficiencia con respecto a la versión serial, lo cual se puede observar en la siguiente gráfica:

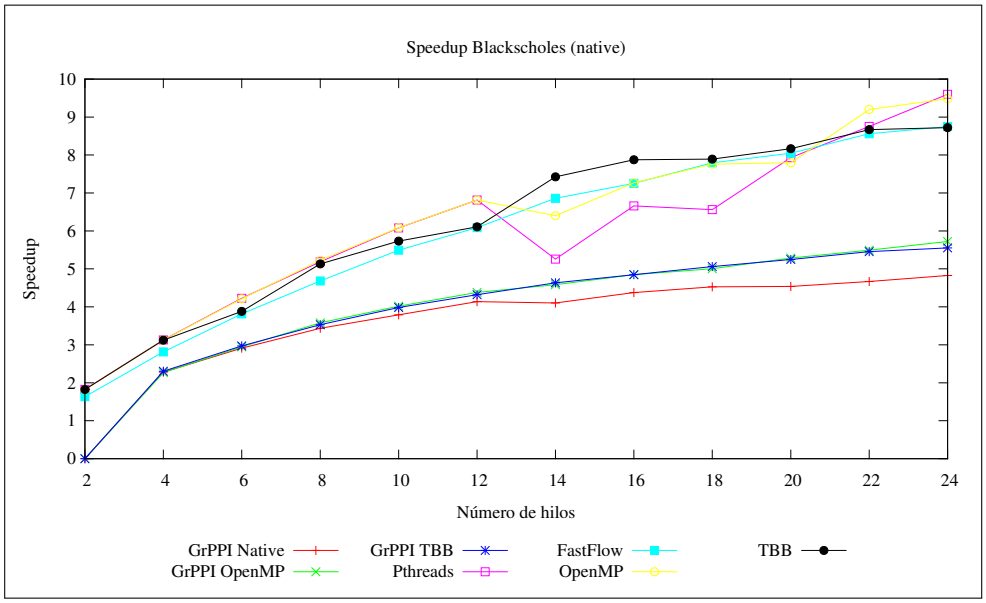


Figura 5-3: *Gráfica de los speedups para los distintos modelos de programación del kernel Blacksholes, usando la implementación de la versión 2 para GrPPI*

Si se analiza la primera gráfica, se puede observar que las líneas correspondientes con *GrPPI Native, GrPPI TBB y GrPPI OpenMP* tienen tiempos de ejecución superiores al resto de modelos, sin embargo, no se puede establecer solo con esta gráfica que modelo de los otros sería el mejor para esta implementación. Por ello se debe analizar la segunda gráfica ya que en ésta se puede ver de manera más clara las diferencias entre la escalabilidad de *TBB, OpenMP, Pthreads y FastFlow*. Todos los modelos tienen altibajos por tanto no hay tanta diferencia entre usar uno u otro, lo que significa que cualquiera de los cuatro modelos mencionados podría ser eficaz en este caso.

Antes de establecer cuál es el mejor modelo para esta aplicación se va a proceder a analizar la versión 2 de la implementación.

Versión 2: Farm anidado con un Map

En esta segunda versión no hay que realizar el análisis del *chunksize* ya que no se usa el patrón *Parallel_for*.

En la siguiente tabla se muestran simplemente las medidas de dispersión relativas a los tiempos de ejecución obtenidos con la versión 2 de la implementación de *GrPPI*, ya que las implementaciones para el resto de los modelos de programación son las mismas y por tanto los tiempos de ejecución son iguales que en el caso anterior.

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
4	*grppi_native*	46008,6	136,9006939	18741,8	$\mu \in$ [45907,89573 ; 46109,30427]
4	*grppi_openmp*	75924,2	258,0759578	66603,2	$\mu \in$ [75734,35909 ; 76114,04091]
4	*grppi_tbb*	46814	145,2084708	21085,5	$\mu \in$ [46707,18452 ; 46920,81548]
6	*grppi_native*	35968,8	317,8933784	101056,2	$\mu \in$ [35734,95734 ; 36202,64266]
6	*grppi_openmp*	47115,4	134,6432323	18128,8	$\mu \in$ [47016,35632 ; 47214,44368]
6	*grppi_tbb*	36170,8	133,6775972	17869,7	$\mu \in$ [36072,46664 ; 36269,13336]
8	*grppi_native*	36812,2	926,1947419	857836,7	$\mu \in$ [36130,89031 ; 37493,50969]
8	*grppi_openmp*	36009,6	104,3733683	10893,8	$\mu \in$ [35932,82286 ; 36086,37714]
8	*grppi_tbb*	30443,6	175,0694148	30649,3	$\mu \in$ [30314,81878 ; 30572,38122]
10	*grppi_native*	27953,8	1587,571636	2520383,7	$\mu \in$ [26785,98087 ; 29121,61913]
10	*grppi_openmp*	30633,4	135,2028106	18279,8	$\mu \in$ [30533,94469 ; 30732,85531]
10	*grppi_tbb*	26945,8	143,6112809	20624,2	$\mu \in$ [26840,15941 ; 27051,44059]
12	*grppi_native*	24912,4	509,2580878	259343,8	$\mu \in$ [24537,78929 ; 25287,01071]
12	*grppi_openmp*	27871,6	186,4143235	34750,3	$\mu \in$ [27734,47346 ; 28008,72654]
12	*grppi_tbb*	24643,4	137,9467288	19029,3	$\mu \in$ [24541,92626 ; 24744,87374]
14	*grppi_native*	23099,6	378,0486741	142920,8	$\mu \in$ [22821,50705 ; 23377,69295]
14	*grppi_openmp*	25292,6	110,757844	12267,3	$\mu \in$ [25211,12643 ; 25374,07357]
14	*grppi_tbb*	24029,8	1543,694497	2382992,7	$\mu \in$ [22894,25693 ; 25165,34307]

16	*grppi_native*	21867,4	209,4523812	43870,3	$\mu \in$ [21713,32664 ; 22021,47336]
16	*grppi_openmp*	23649,2	247,0145745	61016,2	$\mu \in$ [23467,49586 ; 23830,90414]
16	*grppi_tbb*	22818,2	1128,531214	1273582,7	$\mu \in$ [21988,05142 ; 23648,34858]
18	*grppi_native*	21171,6	191,2937532	36593,3	$\mu \in$ [21030,88414 ; 21312,31586]
18	*grppi_openmp*	22687,2	83,02830843	6893,7	$\mu \in$ [22626,1243 ; 22748,2757]
18	*grppi_tbb*	21284	362,971762	131748,5	$\mu \in$ [21016,99764 ; 21551,00236]
20	*grppi_native*	20281	196,7231557	38700	$\mu \in$ [20136,29027 ; 20425,70973]
20	*grppi_openmp*	21362,8	164,5515117	27077,2	$\mu \in$ [21241,75576 ; 21483,84424]
20	*grppi_tbb*	20197,4	159,9102873	25571,3	$\mu \in$ [20079,76985 ; 20315,03015]
22	*grppi_native*	19792,4	117,1251467	13718,3	$\mu \in$ [19706,24264 ; 19878,55736]
22	*grppi_openmp*	21163,6	236,8001267	56074,3	$\mu \in$ [20989,40961 ; 21337,79039]
22	*grppi_tbb*	19597,8	193,2542367	37347,2	$\mu \in$ [19455,64201 ; 19739,95799]
24	*grppi_native*	19327,6	205,3053823	42150,3	$\mu \in$ [19176,57718 ; 19478,62282]
24	*grppi_openmp*	19833,8	136,7029627	18687,7	$\mu \in$ [19733,24118 ; 19934,35882]
24	*grppi_tbb*	19803,4	1613,448109	2603214,8	$\mu \in$ [18616,54611 ; 20990,25389]

Tabla 5.3: *Medidas de dispersión de los tiempos de ejecución obtenidos para la implementación 2 de la aplicación Blackscholes.*

Para entender mejor las medidas establecidas en la tabla anterior y poder hacer una comparación más sencilla con la versión anterior de la implementación, se van a mostrar en la siguiente gráfica tanto las medias para el modelo *GrPPI* de la tabla anterior como las medias del resto de modelos que aparecen en la Tabla 5.2:

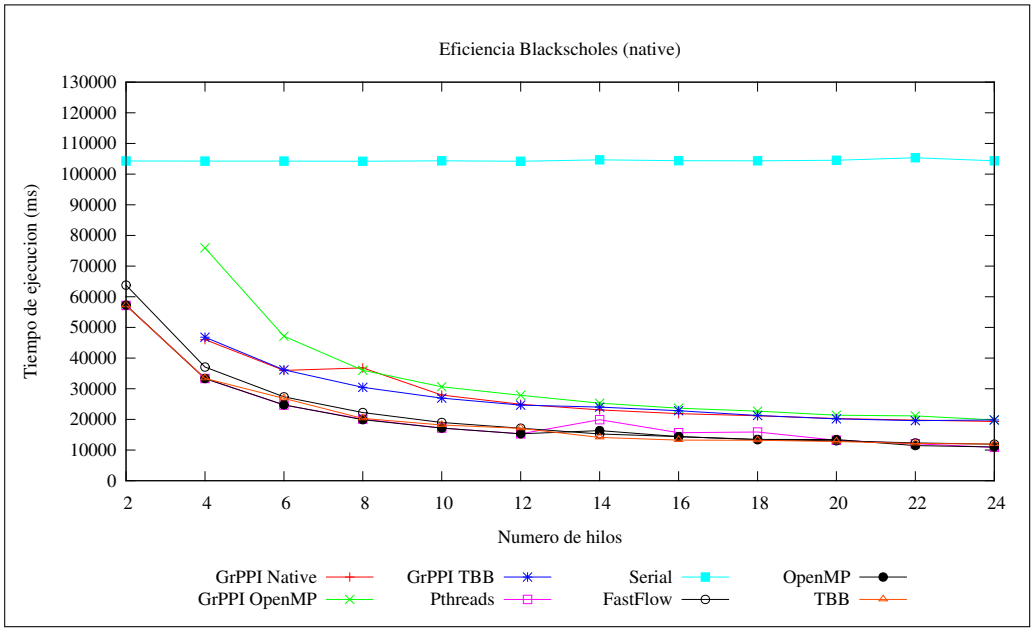


Figura 5-4: *Gráfica de tiempos de ejecución medios para los distintos modelos de programación de la aplicación Blackscholes, usando la implementación de la versión 2 para GrPPI.*

A parte del tiempo de ejecución medio mostrado en la gráfica anterior, también se podría considerar otra medida para la evaluación de los distintos modelos, esta medida es el *speedup* o aumento de la eficiencia con respecto a la versión serial, que se puede observar en la siguiente gráfica:

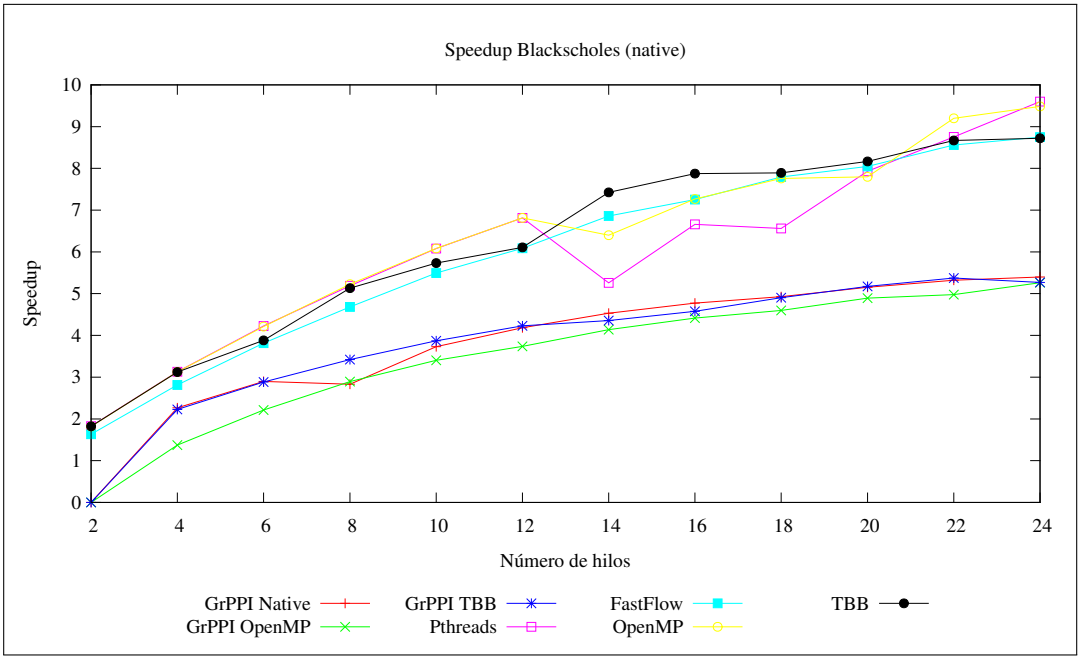


Figura 5-5: *Gráfica de los speedups para los distintos modelos de programación del kernel Blacksholes, usando la implementación de la versión 2 para GrPPI.*

Como se puede comprobar comparando las Gráficas 5-2, 5-4, 5-3 y 5-5 las variaciones entre usar la primera o la segunda implementación de *GrPPI* no son demasiadas, aunque es un poco mejor la primera implementación (*Parallel_for*). Una vez dicho esto se pasaría a analizar únicamente las Gráficas 5-2 y 5-3 para determinar cuál es el mejor modelo en este caso. El mejor modelo podría ser *OpenMP, TBB o FastFlow* como se ha comentado durante el análisis de la implementación anterior, puesto que no es posible elegir un único modelo como el mejor simplemente observando la gráfica anterior.

De forma que se va a pasar a analizar los valores medios del *speedup* para los tres modelos que se han identificado como los mejores.

Modelo	Speedup Medio
TBB	6,213
OpenMP	6,266
GrPPI_OpenMP	5,982

Tabla 5.4: *Speedups medios para los modelos con mejor rendimiento para la aplicación Blackscholes.*

Como se puede observar en la tabla anterior, el modelo con mayor valor medio del *speedup* es *OpenMP* de forma que dicho modelo es el más adecuado para paralelizar esta aplicación.

5.3.2. Bodytrack

En el caso de la aplicación *Bodytrack*, dependiendo del conjunto de entrada que se escoja para la ejecución varía el número de *frames*, partículas y capas o pasadas del filtro de partículas, mientras que el número de cámaras utilizado se mantiene constante en todos los conjuntos de entrada (4 cámaras). En concreto el conjunto de entrada *native* consta de 4 cámaras, 261 *frames*, 4.000 partículas y 5 capas.

En esta aplicación como se ha comentado en la Sección 4.2.2, hay cinco patrones de tipo *Parallel_for* lo que significa que se debe realizar un análisis del *chunksize*, para ello se han realizado varias ejecuciones con 6, 12, 18 y 24 hilos modificando en cada caso el valor del *chunksize* en 20 unidades en un rango de 0 a 460. El resultado de dichas ejecuciones se puede observar en la siguiente gráfica:

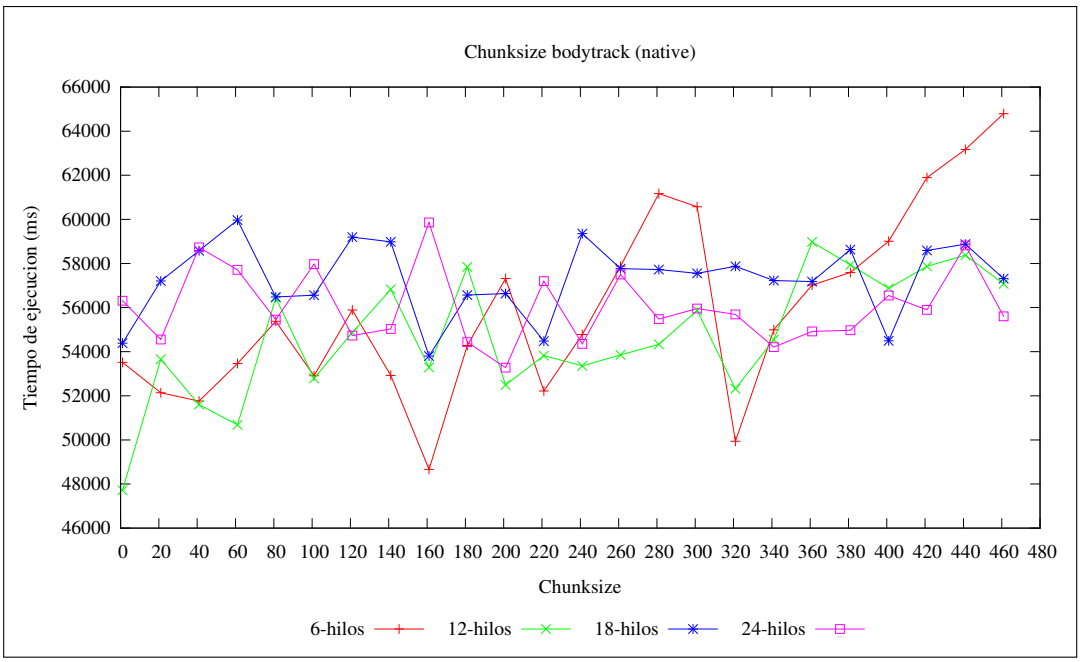


Figura 5-6: *Gráfica de tiempos de ejecución para los distintos valores del chunksize usando la implementación GrPPI Native de la aplicación Bodytrack.*

Como se puede ver en la gráfica anterior los tiempos de ejecución no siguen ningún patrón como pasa en con la aplicación anterior y por tanto no es sencillo determinar cuál sería el mejor valor. Dados los datos del problema y los resultados observado se ha optado por establecer 1 como el valor final del *chunksize* con el que realizar el resto de las ejecuciones.

A continuación, se muestra la tabla con los resultados de las medidas de dispersión aplicadas sobre los tiempos de ejecución obtenidos para cada uno de los modelos:

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
1	serial	120735,2	864,617372	747563,2	$\mu \in$ [120099,1867 ; 240834,3867]

2	fastflow	97204,6	305,3126267	93215,8	$\mu \in$ [96980,01176 ; 97429,18824]
2	pthreads	85281	5674,933127	32204866	$\mu \in$ [81106,51406 ; 89455,48594]
2	openmp	69678,4	177,8153537	31618,3	$\mu \in$ [69547,59886 ; 69809,20114]
2	tbb	69992	397,7807185	158229,5	$\mu \in$ [69699,39214 ; 70284,60786]
2	*grppi_native*	94600,6	665,4508246	442824,8	$\mu \in$ [94111,09377 ; 95090,10623]
2	*grppi_openmp*	71454,8	563,7142006	317773,7	$\mu \in$ [71040,13132 ; 71869,46868]
4	fastflow	60183	424,3671759	180087,5	$\mu \in$ [59870,83512 ; 60495,16488]
4	pthreads	48963,6	622,021945	386911,3	$\mu \in$ [48506,04009 ; 49421,15991]
4	openmp	49473,2	371,5477089	138047,7	$\mu \in$ [49199,88917 ; 49746,51083]
4	tbb	48565,2	726,0032369	527080,7	$\mu \in$ [48031,15136 ; 49099,24864]
4	*grppi_native*	60713,4	667,6034002	445694,3	$\mu \in$ [60222,31033 ; 61204,48967]
4	*grppi_openmp*	51453,6	835,5036206	698066,3	$\mu \in$ [50839,00278 ; 52068,19722]
6	fastflow	48704,8	436,4019936	190446,7	$\mu \in$ [48383,7823 ; 49025,8177]
6	pthreads	34390	696,432337	485018	$\mu \in$ [33877,70374 ; 34902,29626]
6	openmp	35906,4	320,0442157	102428,3	$\mu \in$ [35670,97519 ; 36141,82481]
6	tbb	36480	511,34284	261471,5	$\mu \in$ [36103,85574 ; 36856,14426]
6	*grppi_native*	52981	848,1120209	719294	$\mu \in$ [52357,12803 ; 53604,87197]
6	*grppi_openmp*	36164,2	1820,170651	3313021,2	$\mu \in$ [34825,28082 ; 37503,11918]
8	fastflow	42814,4	377,151561	142243,3	$\mu \in$ [42536,96697 ; 43091,83303]
8	pthreads	28428,8	689,4267184	475309,2	$\mu \in$ [27921,65708 ; 28935,94292]
8	openmp	29106,8	503,4502955	253462,2	$\mu \in$ [28736,46151 ; 29477,13849]
8	tbb	27484	246,4528352	60739	$\mu \in$ [27302,70907 ; 27665,29093]
8	*grppi_native*	53072,6	3050,47493	9305397,3	$\mu \in$ [50828,66788 ; 55316,53212]
8	*grppi_openmp*	30022,2	783,1766723	613365,7	$\mu \in$ [29446,09453 ; 30598,30547]
10	fastflow	38681,6	394,358086	155518,3	$\mu \in$ [38391,50984 ; 38971,69016]
10	pthreads	24326,2	973,8293998	948343,7	$\mu \in$ [23609,85021 ; 25042,54979]
10	openmp	25265,2	844,6346547	713407,7	$\mu \in$ [24643,88598 ; 25886,51402]
10	tbb	24253	636,1147695	404642	$\mu \in$ [23785,0734 ; 24720,9266]
10	*grppi_native*	51383,2	3093,724244	9571129,7	$\mu \in$ [49107,45365 ; 53658,94635]
10	*grppi_openmp*	25787,2	764,2883618	584136,7	$\mu \in$ [25224,98879 ; 26349,41121]
12	fastflow	37755	272,7159695	74374	$\mu \in$ [37554,38989 ; 37955,61011]
12	pthreads	21750,6	771,6853633	595498,3	$\mu \in$ [21182,94755 ; 22318,25245]
12	openmp	22169,8	434,7053025	188968,7	$\mu \in$ [21850,03039 ; 22489,56961]
12	tbb	22549	1043,911634	1089751,5	$\mu \in$ [21781,09766 ; 23316,90234]
12	*grppi_native*	49305	2650,228198	7023709,5	$\mu \in$ [47355,48974 ; 51254,51026]
12	*grppi_openmp*	22580,6	246,1357349	60582,8	$\mu \in$ [22399,54233 ; 22761,65767]
14	fastflow	16292,4	210,6983626	44393,8	$\mu \in$ [16137,41009 ; 16447,38991]

14	pthreads	24289	835,9910287	698881	$\mu \in$ [23674,04424 ; 24903,95576]
14	openmp	22032,4	453,4746961	205639,3	$\mu \in$ [21698,8236 ; 22365,9764]
14	tbb	21771,2	931,9089011	868454,2	$\mu \in$ [21085,68697 ; 22456,71303]
14	*grppi_native*	66043,8	1488,11431	2214484,2	$\mu \in$ [64949,14177 ; 67138,45823]
14	*grppi_openmp*	2368,6	935,0913324	874395,8	$\mu \in$ [21680,74597 ; 23056,45403]
16	fastflow	32977,2	154,2212696	23784,2	$\mu \in$ [32863,75469 ; 33090,64531]
16	pthreads	24782	712,9460008	508292	$\mu \in$ [24257,55628 ; 25306,44372]
16	openmp	20985	760,011842	577618	$\mu \in$ [20425,9346 ; 21544,0654]
16	tbb	19866	935,9051768	875918,5	$\mu \in$ [19177,54731 ; 20554,45269]
16	*grppi_native*	57429,6	4047,465911	16381980,3	$\mu \in$ [54452,28041 ; 60406,91959]
16	*grppi_openmp*	20686	681,9021191	464990,5	$\mu \in$ [20184,39218 ; 21187,60782]
18	fastflow	32113	158,202402	25028	$\mu \in$ [31996,62617 ; 32229,37383]
18	pthreads	23874,2	732,399959	536409,7	$\mu \in$ [23335,44593 ; 24412,95407]
18	openmp	19267,8	448,5149942	201165,7	$\mu \in$ [18937,87196 ; 19597,72804]
18	tbb	19120,6	1024,918192	1050457,3	$\mu \in$ [18366,66925 ; 19874,53075]
18	*grppi_native*	55128,6	2195,586323	4820599,3	$\mu \in$ [53513,52472 ; 56743,67528]
18	*grppi_openmp*	19955	703,2890586	494615,5	$\mu \in$ [19437,65993 ; 20472,34007]
20	fastflow	30968,8	137,5452653	18918,7	$\mu \in$ [30867,62158 ; 31069,97842]
20	pthreads	23131,8	294,9867794	87017,2	$\mu \in$ [22914,80746 ; 23348,79254]
20	openmp	18928,6	433,4908304	187914,3	$\mu \in$ [18609,72375 ; 19247,47625]
20	tbb	18444,4	545,427172	297490,8	$\mu \in$ [18043,18328 ; 18845,61672]
20	*grppi_native*	55017,6	2466,777817	6084992,8	$\mu \in$ [53203,03601 ; 56832,16399]
20	*grppi_openmp*	18887,4	328,8119523	108117,3	$\mu \in$ [18645,52563 ; 19129,27437]
22	fastflow	30244,6	1225,261931	1501266,8	$\mu \in$ [29343,29622 ; 31145,90378]
22	pthreads	22898	364,6621724	132978,5	$\mu \in$ [22629,75418 ; 23166,24582]
22	openmp	17880,8	284,8327579	81129,7	$\mu \in$ [17671,27677 ; 18090,32323]
22	tbb	17756,8	642,2937023	412541,2	$\mu \in$ [17284,32817 ; 18229,27183]
22	*grppi_native*	55572,2	3014,290995	9085950,2	$\mu \in$ [53354,88482 ; 57789,51518]
22	*grppi_openmp*	18383,8	1096,051413	1201328,7	$\mu \in$ [17577,54359 ; 19190,05641]
24	fastflow	30321,6	83,61399404	6991,3	$\mu \in$ [30260,09347 ; 30383,10653]
24	pthreads	22444,2	440,0593142	193652,2	$\mu \in$ [22120,49197 ; 22767,90803]
24	openmp	16995,6	409,2973247	167524,3	$\mu \in$ [16694,52052 ; 17296,67948]
24	tbb	17087,6	564,6687525	318850,8	$\mu \in$ [16672,22915 ; 17502,97085]
24	*grppi_native*	55389,2	1215,613302	1477715,7	$\mu \in$ [54494,99376 ; 56283,40624]
24	*grppi_openmp*	17490,6	328,1330523	107671,3	$\mu \in$ [17249,22503 ; 17731,97497]

Tabla 5.5: *Medidas de dispersión de los tiempos de ejecución obtenidos para la aplicación Bodytrack.*

A continuación, se muestran las medias resultantes de los tiempos de ejecución expuestas en la tabla anterior en forma de gráfica para una mejor evaluación:

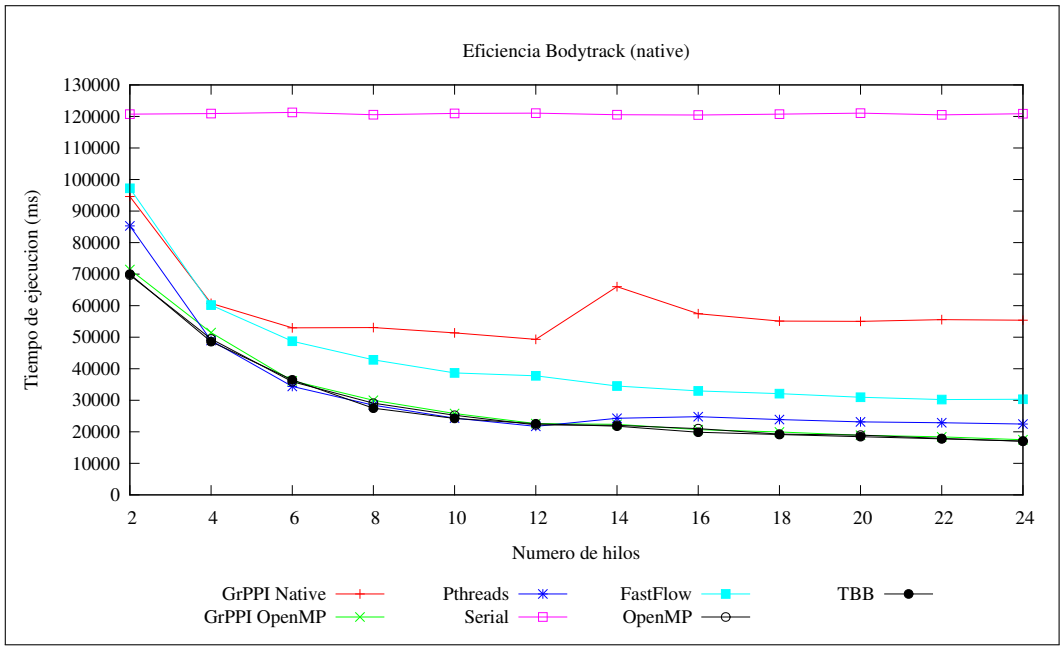


Figura 5-7: *Gráfica de tiempos de ejecución medios para los distintos modelos de programación de la aplicación Bodytrack.*

A parte del tiempo de ejecución medio, también se podría considerar otra medida para la evaluación de los modelos, esta medida es el *speedup* o aumento de la eficiencia con respecto a la versión serial, lo cual se puede observar en la siguiente gráfica:

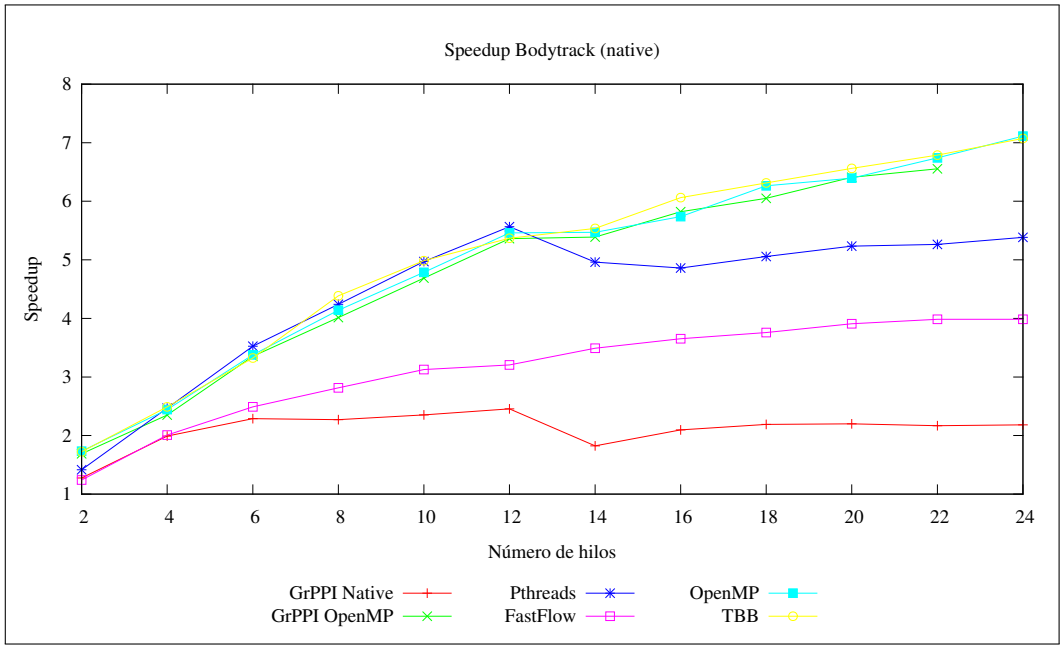


Figura 5-8: *Gráfica de los speedups para los distintos modelos de programación de la aplicación Bodytrack.*

Analizando la primera gráfica, se puede observar que los tiempos de ejecución medios para los modelos *GrPPI Native y FastFlow* son superiores a los del resto de modelos. En cuanto al resto de modelos, los tiempos están bastante igualados, por tanto, se debe pasar a analizar la segunda gráfica. En ésta se ve como el modelo *Pthreads* a pesar de tener una escalabilidad equivalente a los modelos *OpenMP, TBB y GrPPI OpenMP* en la primera mitad de la gráfica, en la segunda mitad el *speedup* comienza a disminuir. Teniendo en cuenta todo este análisis y el hecho de que los tres modelos restantes tienen *speedups* equivalentes, no se puede elegir un único modelo como el mejor simplemente mirando la gráfica.

De forma que se debe pasar realizar un análisis basado en los valores obtenidos para el *speedup* usando esos tres modelos que se consideran mejores:

Modelo	Speedup Medio
TBB	5,051
OpenMP	4,972
GrPPI_OpenMP	4,307

Tabla 5.6: *Speedups medios para los modelos con mejor rendimiento para la aplicación Bodytrack.*

Como se puede observar en la tabla anterior, el modelo con mayor valor medio del *speedup* es *TBB* de forma que dicho modelo es el más adecuado para paralelizar esta aplicación.

5.3.3. Canneal

En el caso del kernel *Canneal*, dependiendo del conjunto de entrada se modifica el número de cambios (swaps) por unidad de temperatura, la temperatura inicial y el número de elementos de tipo *netlist*. En el caso del conjunto *native* escogido los valores de dichos parámetros son: 15.000 *swaps* por unidad de temperatura, 2.000 grados de temperatura inicial y 2.500.000 elementos de tipo *netlist*.

Como este kernel utiliza el patrón *Parallel_for* como se ha comentado en la Sección 4.2.3, el primer paso para la evaluación consistió en hacer un análisis del *chunksize* para determinar cuál era el valor más adecuado para hacer las divisiones de los datos necesarias para la versión *Native* de *GrPPI* en este caso. Se ejecutó el programa utilizando 6, 12, 18 y 24 hilos, teniendo en cuenta que en cada caso el valor del *chunksize* va desde el 1 hasta el número de hilos máximo para ese caso, es decir, en la prueba con 6 hilos se ejecutó con los valores 1, 2, 3, 4, 5 y 6. Como resultado de estas ejecuciones se obtuvo la siguiente gráfica:

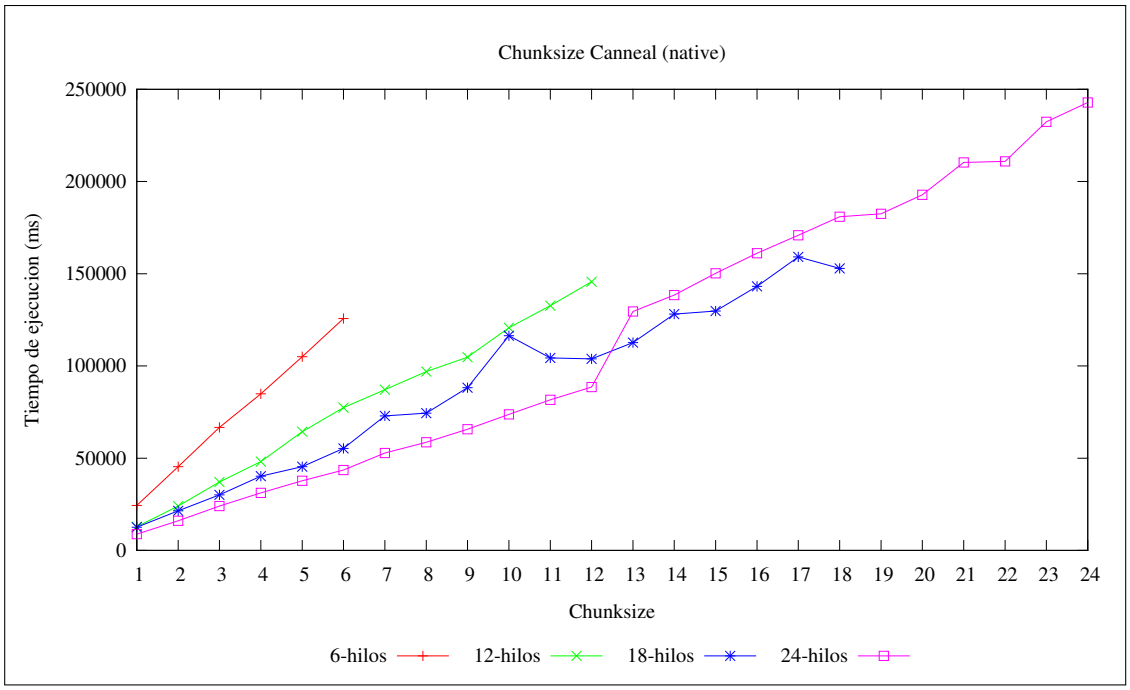


Figura 5-9: *Gráfica de los tiempos de ejecución para los distintos valores del chunksize obtenidos para la versión GrPPI Native del kernel Canneal.*

Como se puede observar en la gráfica anterior, el valor con menor tiempo de ejecución para todos los casos es el 1, por ello este fue el valor escogido para realizar el resto de las ejecuciones del programa.

Una vez seleccionado dicho valor, se procedió a realizar las ejecuciones para la posterior evaluación de los tiempos obtenidos para cada uno de los modelos. Se han realizado varias ejecuciones del programa con los distintos modelos de programación a evaluar, variando para cada uno de ellos el número de hilos utilizado, de forma que se pudiera ver la evolución del tiempo de ejecución con dichos cambios.

A continuación, se muestra una tabla con las medidas de dispersión (media, desviación estándar, varianza e intervalo de confianza) de los tiempos extraídos para cada uno de los modelos de programación, mencionados en la Tabla 4.3:

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
1	serial	79084,2	1629,198944	2654289,2	$\mu \in$ [77885,75978 ; 80282,6402]
2	fastflow	72255	5145,138628	26472451,5	$\mu \in$ [68470,23137 ; 76039,76863]
2	pthreads	74397,2	4481,842333	20086910,7	$\mu \in$ [71100,35273 ; 77694,04727]
2	*grppi_native*	74312,8	3735,310389	13952543,7	$\mu \in$ [71565,1023 ; 77060,4977]
2	*grppi_openmp*	66492,6	1441,426134	2077709,3	$\mu \in$ [65432,28563 ; 67552,91437]
2	*grppi_tbb*	66309,8	1268,946492	1610225,2	$\mu \in$ [65376,36181 ; 67243,23819]
4	fastflow	38240,6	2359,926863	5569254,8	$\mu \in$ [36504,63566 ; 39976,56434]
4	pthreads	37831,8	2571,619956	6613229,2	$\mu \in$ [35940,11403 ; 39723,48597]

4	*grppi_native*	36854,4	2052,774293	4213882,3	$\mu \in$ [35344,37737 ; 38364,42263]
4	*grppi_openmp*	34623,2	640,8577845	410698,7	$\mu \in$ [34151,78443 ; 35094,61557]
4	*grppi_tbb*	35266,2	808,5290347	653719,2	$\mu \in$ [34671,44531 ; 35860,95469]
6	fastflow	25973,8	1362,60218	1856684,7	$\mu \in$ [24971,4686 ; 26976,1314]
6	pthreads	25554	2026,918844	4108400	$\mu \in$ [24062,99666 ; 27045,00334]
6	*grppi_native*	27497,8	27,43537862	752,7	$\mu \in$ [27477,61851 ; 27517,98149]
6	*grppi_openmp*	24099	526,2860439	276977	$\mu \in$ [23711,86351 ; 24486,13649]
6	*grppi_tbb*	23736,6	443,0285544	196274,3	$\mu \in$ [23410,70779 ; 24062,49221]
8	fastflow	20256,6	1136,483524	1291594,8	$\mu \in$ [19420,60169 ; 21092,59831]
8	pthreads	18206	23,80126047	566,5	$\mu \in$ [18188,49177 ; 18223,50823]
8	*grppi_native*	19564,8	955,1035022	912222,7	$\mu \in$ [18862,225 ; 20267,375]
8	*grppi_openmp*	17966,6	36,40467003	1325,3	$\mu \in$ [17939,82069 ; 17993,37931]
8	*grppi_tbb*	17945,4	30,71318935	943,3	$\mu \in$ [17922,80735 ; 17967,99265]
10	fastflow	17018	79,52043762	6323,5	$\mu \in$ [16959,50469 ; 17076,49531]
10	pthreads	14952,2	272,8528908	74448,7	$\mu \in$ [14751,48917 ; 15152,91083]
10	*grppi_native*	16715	78,43787351	6152,5	$\mu \in$ [16657,30103 ; 16772,69897]
10	*grppi_openmp*	14950,2	330,3750899	109147,7	$\mu \in$ [14707,17579 ; 5193,22421]
10	*grppi_tbb*	16594,6	69,91995423	4888,8	$\mu \in$ [16543,16682 ; 16646,03318]
12	fastflow	17804,2	173,8899077	30237,7	$\mu \in$ [17676,28643 ; 17932,11357]
12	pthreads	12706	227,702657	51848,5	$\mu \in$ [12538,50172 ; 12873,49828]
12	*grppi_native*	13343,4	625,7162296	391520,8	$\mu \in$ [12883,12258 ; 13803,67742]
12	*grppi_openmp*	12678,4	264,7665009	70101,3	$\mu \in$ [12483,63752 ; 12873,16248]
12	*grppi_tbb*	13298,2	633,5275842	401357,2	$\mu \in$ [12832,17654 ; 13764,22346]
14	fastflow	16292,4	210,6983626	44393,8	$\mu \in$ [16137,41009 ; 16447,38991]
14	pthreads	17032,8	216,863321	47029,7	$\mu \in$ [16873,27514 ; 17192,32486]
14	*grppi_native*	15383,2	1723,899272	2971828,7	$\mu \in$ [14115,09814 ; 16651,30186]
14	*grppi_openmp*	14724,6	693,4975126	480938,8	$\mu \in$ [14214,4626 ; 15234,7374]
14	*grppi_tbb*	16965,2	2036,908736	4148997,2	$\mu \in$ [15466,84809 ; 18463,55191]
16	fastflow	14118,8	498,6568961	248658,7	$\mu \in$ [13751,98754 ; 14485,61246]
16	pthreads	14693,8	377,7091474	142664,2	$\mu \in$ [14415,95681 ; 14971,64319]
16	*grppi_native*	13265	1272,770796	1619945,5	$\mu \in$ [12328,74865 ; 14201,25135]
16	*grppi_openmp*	12854,2	732,3692375	536364,7	$\mu \in$ [12315,46853 ; 13392,93147]
16	*grppi_tbb*	14054,4	1813,795826	3289855,3	$\mu \in$ [12720,17015 ; 15388,62985]
18	fastflow	12947,8	363,1393672	131870,2	$\mu \in$ [12680,67435 ; 13214,92565]
18	pthreads	13359,6	51,51989907	2654,3	$\mu \in$ [13321,70192 ; 13397,49808]
18	*grppi_native*	11831,6	921,1570442	848530,3	$\mu \in$ [11153,99605 ; 12509,20395]

18	*grppi_openmp*	12714,2	1124,58379	1264688,7	$\mu \in$ [11886,95515 ; 13541,44485]
18	*grppi_tbb*	12604,6	1702,951056	2900042,3	$\mu \in$ [11351,90766 ; 13857,29234]
20	fastflow	11767,6	322,4551132	103977,3	$\mu \in$ [11530,40173 ; 12004,79827]
20	pthreads	11690,6	428,0926302	183263,3	$\mu \in$ [11375,69467 ; 12005,50533]
20	*grppi_native*	10738,4	750,8403958	563761,3	$\mu \in$ [10186,08113 ; 11290,71887]
20	*grppi_openmp*	10678,8	818,6285482	670152,7	$\mu \in$ [10076,6161 ; 11280,9839]
20	*grppi_tbb*	11117,6	771,2838647	594878,8	$\mu \in$ [10550,24289 ; 11684,95711]
22	fastflow	10968,2	172,9239717	29902,7	$\mu \in$ [10840,99697 ; 11095,40303]
22	pthreads	11201,6	278,491113	77557,3	$\mu \in$ [10996,74169 ; 11406,45831]
22	*grppi_native*	9482,8	133,6027694	17849,7	$\mu \in$ [9384,521682 ; 9581,078318]
22	*grppi_openmp*	9510,2	102,2384468	10452,7	$\mu \in$ [9434,993306 ; 9585,406694]
22	*grppi_tbb*	9451	124,6735738	15543,5	$\mu \in$ [9359,290006 ; 9542,709994]
24	fastflow	10961,4	207,8023099	43181,8	$\mu \in$ [10808,54043 ; 11114,25957]
24	pthreads	10369,4	222,7561447	49620,3	$\mu \in$ [10205,54038 ; 10533,25962]
24	*grppi_native*	8840,8	104,1114787	10839,2	$\mu \in$ [8764,215502 ; 8917,384498]
24	*grppi_openmp*	8770	138,517147	19187	$\mu \in$ [8668,106661 ; 8871,893339]
24	*grppi_tbb*	8805,6	80,29819425	6447,8	$\mu \in$ [8746,532576 ; 8864,667424]

Tabla 5.7: *Medidas de dispersión de los tiempos de ejecución obtenidos para el kernel Canneal.*

Como se puede ver en la tabla anterior, el uso de modelos de programación paralela supone una gran mejora con respecto a la versión secuencial, por ejemplo, en el caso concreto de usar 24 hilos se pasaría de un tiempo de ejecución de 79 segundos a uno de 8 segundos para la versión paralela usando *GrPPI*, lo que supone una disminución del tiempo de ejecución casi del 90 %.

A continuación, se muestran esos mismos datos mediante una gráfica para una mejor comparativa de los distintos modelos paralelos:

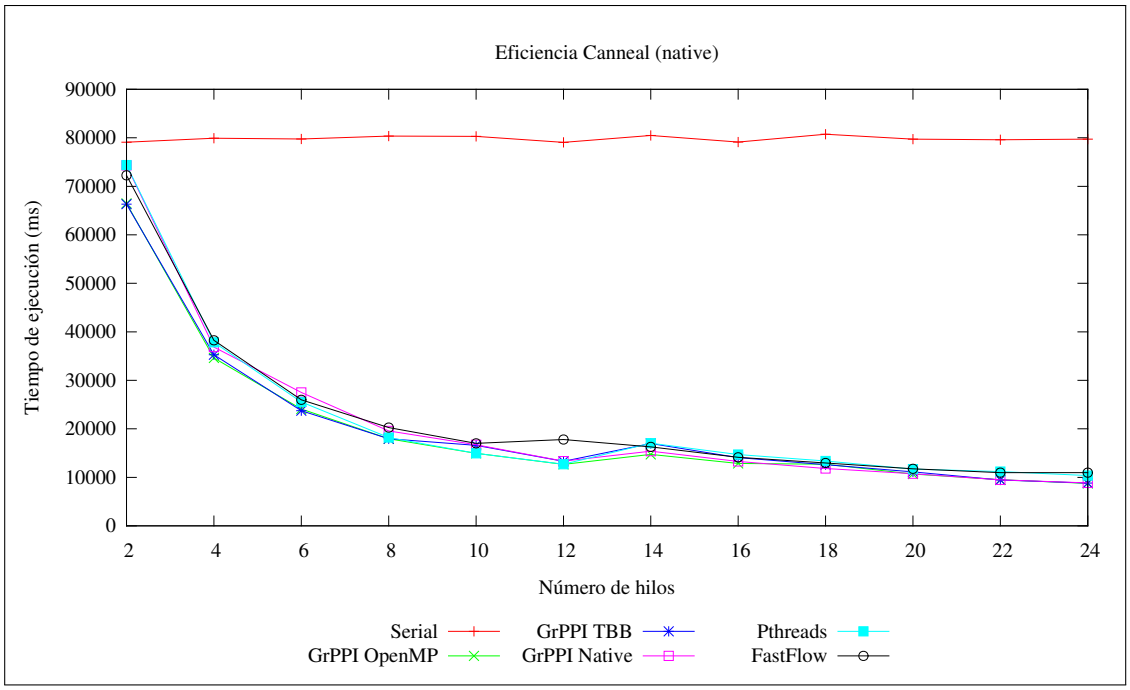


Figura 5-10: *Gráfica de tiempos de ejecución medios para los distintos modelos de programación del kernel Canneal.*

A parte del tiempo de ejecución medio se ha considerado el *speedup* o aumento de la eficiencia con respecto a la versión serial para realizar la evaluación, cuyos valores se pueden observar en la siguiente gráfica:

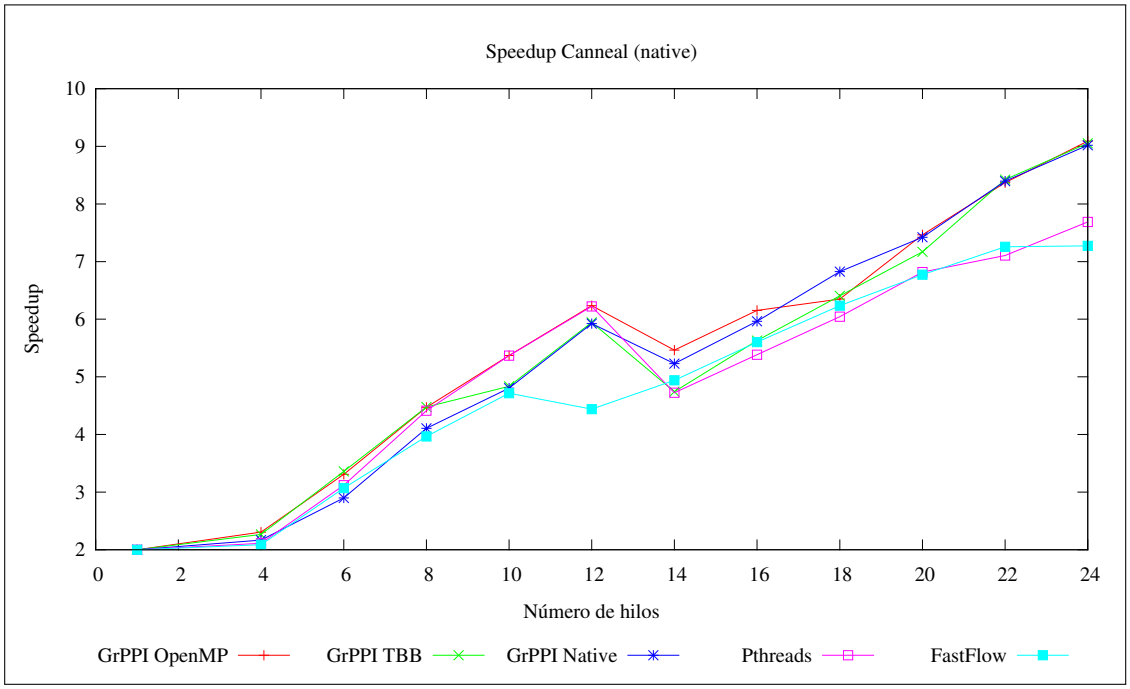


Figura 5-11: *Gráfica de los speedups para los distintos modelos de programación del kernel Canneal.*

Aunque en la primera gráfica (Figura 5-10) parece que los tiempos de ejecución son muy similares, en

la segunda (Figura 5-11) se pueden ver las ligeras variaciones entre los distintos modelos. En la primera sección de la gráfica (rango entre 0 y 12) los modelos con mayor escalabilidad en todos los casos son *GrPPI OpenMP y Pthreads* seguidos por los modelos *GrPPI TBB y GrPPI Native*, sin embargo, en la segunda parte de ésta la eficiencia del modelo *Pthreads* disminuye notablemente lo que provoca que el resto de los modelos le sobrepasen. En general, el modelo con un *speedup* más establece durante toda la gráfica es *GrPPI OpenMP*, por lo que se podría considerar como el mejor modelo para este *kernel*.

5.3.4. Facesim

En el caso de la aplicación *Facesim*, dependiendo del conjunto de entrada que se escoja se podrían modificar el número de partículas, número de tetraedros y el número de *frames*, sin embargo, en todos los conjuntos de entrada de este programa se mantienen constantes el número de partículas (80.598) y el número de tetraedros (372.126), por lo tanto, lo único que varía entre los distintos conjuntos es el número de *frames*. En concreto el conjunto *native* usa 100 *frames* en el cálculo.

Esta aplicación usa varias veces el patrón *Parallel_for* de forma que es necesario hacer un análisis para determinar el mejor valor para el *chunksize*. Se han realizado ejecuciones usando 2, 6, 8 y 16 hilos variando el *chunksize* en 1 unidad en el rango de 1 al número de hilos y se han obtenido los siguientes resultados.

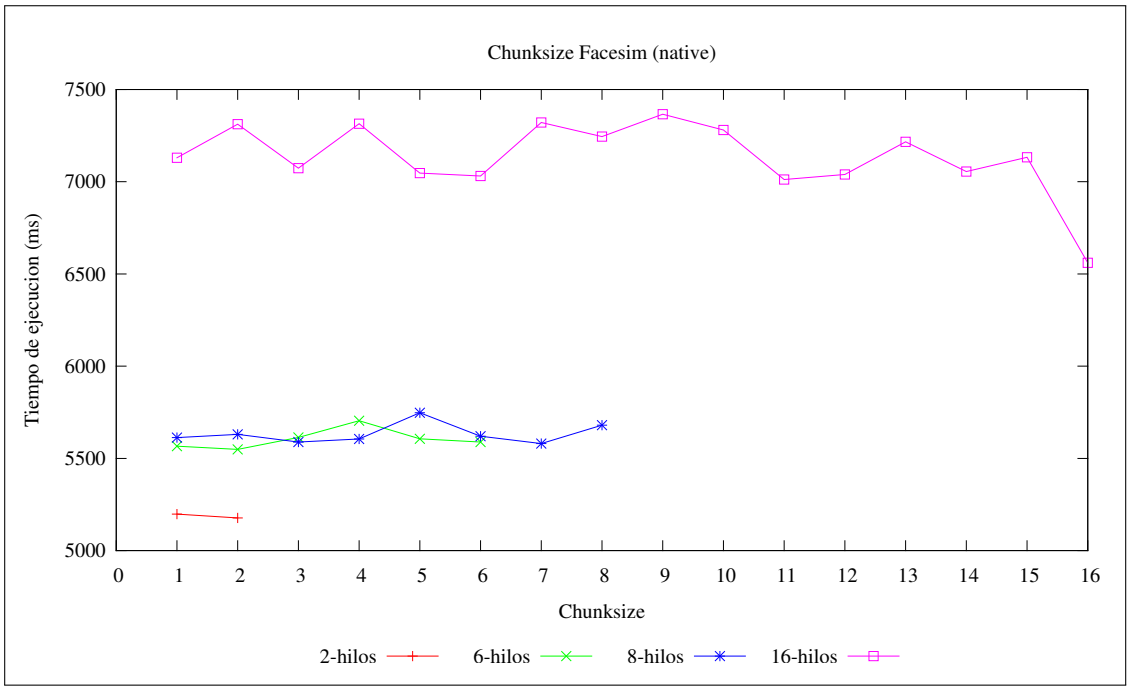


Figura 5-12: *Gráfica de tiempos de ejecución para los distintos valores del chunksize obtenidos mediante la implementación GrPPI Native de la aplicación Facesim.*

Como se puede ver los valores obtenidos para las ejecuciones con 2 hilos son menores que los resultantes de las ejecuciones con 6, 8 y 16 hilos, esto es debido a que dependiendo del número de hilos que se establezca se coge un archivo de entrada u otro, de forma que los archivos destinados a un mayor número de hilos tienen más carga de trabajo lo que conlleva un aumento en sus tiempos de ejecución. Teniendo en cuenta la distribución que se muestra en la gráfica anterior, se decidió usar 1 como el valor del *chunksize* para el resto de las ejecuciones.

A continuación, se muestra la tabla con las medidas de dispersión obtenidas a partir de las ejecuciones. Esta aplicación tiene una restricción con respecto al número de hilos que se pueden usar, los valores válidos son 1, 2, 3, 4, 6, 8, 16, 32, 64 y 128, aunque en este caso solo se han usado los valores en el rango de 1 a 16 ya que se ha limitado el número de hilos de los experimentos al número de núcleos de los que consta la maquina sobre la que se llevan a cabo las ejecuciones (24 núcleos).

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
1	serial	367317,6	1006,560877	1013164,8	$\mu \in$ [366577,1729 ; 368058,0271]
1	fastflow	364626,6	2821,181012	7959062,3	$\mu \in$ [362551,3367 ; 366701,8633]
1	pthreads	372393	4284,020191	18352829	$\mu \in$ [369241,6709 ; 375544,3291]
1	*grppi_native*	370300,4	3469,255511	12035733,8	$\mu \in$ [367748,4125 ; 372852,3875]
1	*grppi_openmp*	347523,4	1234,359064	1523642,3	$\mu \in$ [346615,4044 ; 348431,3956]
1	*grppi_tbb*	372291,4	2536,036435	6431480,8	$\mu \in$ [370425,8893 ; 374156,9107]
2	fastflow	192503,6	1095,045342	1199124,3	$\mu \in$ [191698,0837 ; 193309,1163]
2	pthreads	191440,6	2030,846203	4124336,3	$\mu \in$ [189946,7077 ; 192934,4923]
2	*grppi_native*	193049,6	1847,087654	3411732,8	$\mu \in$ [191690,8807 ; 194408,3193]
2	*grppi_openmp*	183355,2	1668,155179	2782741,7	$\mu \in$ [182128,1035 ; 184582,2965]
2	*grppi_tbb*	193043,6	3463,687962	11997134,3	$\mu \in$ [190495,708 ; 195591,492]
3	fastflow	138518,2	1425,941514	2033309,2	$\mu \in$ [137469,2761 ; 139567,1239]
3	pthreads	141120	2515,007256	6325261,5	$\mu \in$ [139269,9584 ; 142970,0416]
3	*grppi_native*	139857	1565,019648	2449286,5	$\mu \in$ [138705,7701 ; 141008,2299]
3	*grppi_openmp*	130605,2	1289,070479	1661702,7	$\mu \in$ [129656,9586 ; 131553,4414]
3	*grppi_tbb*	138680,6	960,4193355	922405,3	$\mu \in$ [137974,1147 ; 139387,0853]
4	fastflow	110859,2	843,0890819	710799,2	$\mu \in$ [110239,0229 ; 111479,3771]
4	pthreads	112605,2	1820,05184	3312588,7	$\mu \in$ [111266,3682 ; 113944,0318]
4	*grppi_native*	113129,8	790,8790679	625489,7	$\mu \in$ [112548,0286 ; 113711,5714]
4	*grppi_openmp*	104485,8	1012,570343	1025298,7	$\mu \in$ [103740,9523 ; 105230,6477]
4	*grppi_tbb*	110230,6	1402,806758	1967866,8	$\mu \in$ [109198,6941 ; 111262,5059]
6	fastflow	89969,6	5355,777843	28684356,3	$\mu \in$ [86029,88497 ; 93909,31503]
6	pthreads	95792,8	818,1049444	669295,7	$\mu \in$ [95191,00126 ; 96394,59874]

6	*grppi_native*	111368,8	19094,91631	364615828,7	$\mu \in$ [97322,56229 ; 125415,0377]
6	*grppi_openmp*	75977,4	626,1695457	392088,3	$\mu \in$ [75516,78912 ; 76438,01088]
6	*grppi_tbb*	79641,2	683,4553387	467111,2	$\mu \in$ [79138,44963 ; 80143,95037]
8	fastflow	73365,4	620,8762357	385487,3	$\mu \in$ [72908,68288 ; 73822,11712]
8	pthreads	76077,4	716,272504	513046,3	$\mu \in$ [75550,5093 ; 76604,2907]
8	*grppi_native*	100408,4	24560,86269	603235976,3	$\mu \in$ [82341,40718 ; 118475,3928]
8	*grppi_openmp*	59160,8	477,2391434	227757,2	$\mu \in$ [58809,74245 ; 59511,85755]
8	*grppi_tbb*	62256,8	851,2964818	724705,7	$\mu \in$ [61630,58554 ; 62883,01446]
16	fastflow	51939,2	712,4322424	507559,7	$\mu \in$ [51415,1342 ; 52463,2658]
16	pthreads	91335,4	1297,415238	1683286,3	$\mu \in$ [90381,02018 ; 92289,77982]
16	*grppi_native*	174760,8	12073,14071	145760726,7	$\mu \in$ [165879,7868 ; 183641,8132]
16	*grppi_openmp*	39952,6	3892,326656	15150206,8	$\mu \in$ [37089,40099 ; 42815,79901]
16	*grppi_tbb*	38544,6	1584,43501	2510434,3	$\mu \in$ [37379,08817 ; 39710,11183]

Tabla 5.8: *Medidas de dispersión de los tiempos de ejecución obtenidos para la aplicación Facesim.*

En la siguiente gráfica se muestran los mismos tiempos de ejecución medios que en la tabla anterior con el objetivo de tener una visión más clara de los resultados y poder realizar la evaluación de forma más sencilla.

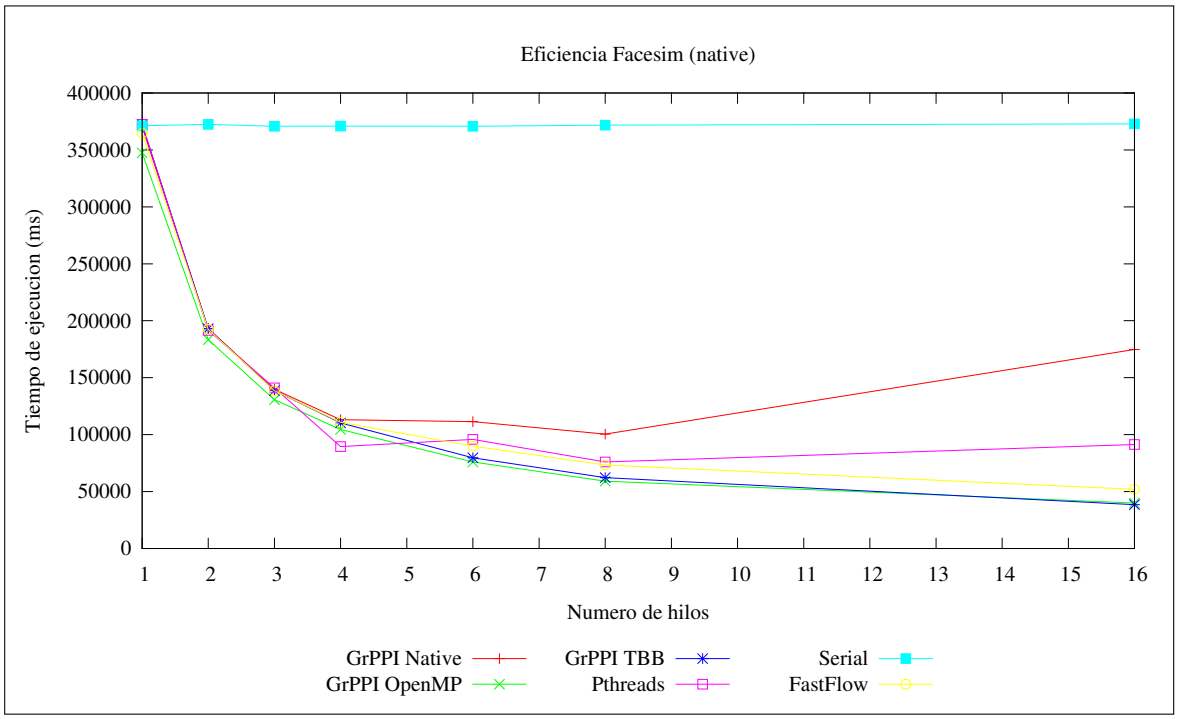


Figura 5-13: *Gráfica de tiempos de ejecución medios para la primera versión de la implementación de la aplicación Facesim.*

A parte del tiempo de ejecución medio, también se podría considerar otra medida para la evaluación de los modelos, esta medida es el *speedup* o aumento de la eficiencia con respecto a la versión serial, lo cual se puede observar en la siguiente gráfica:

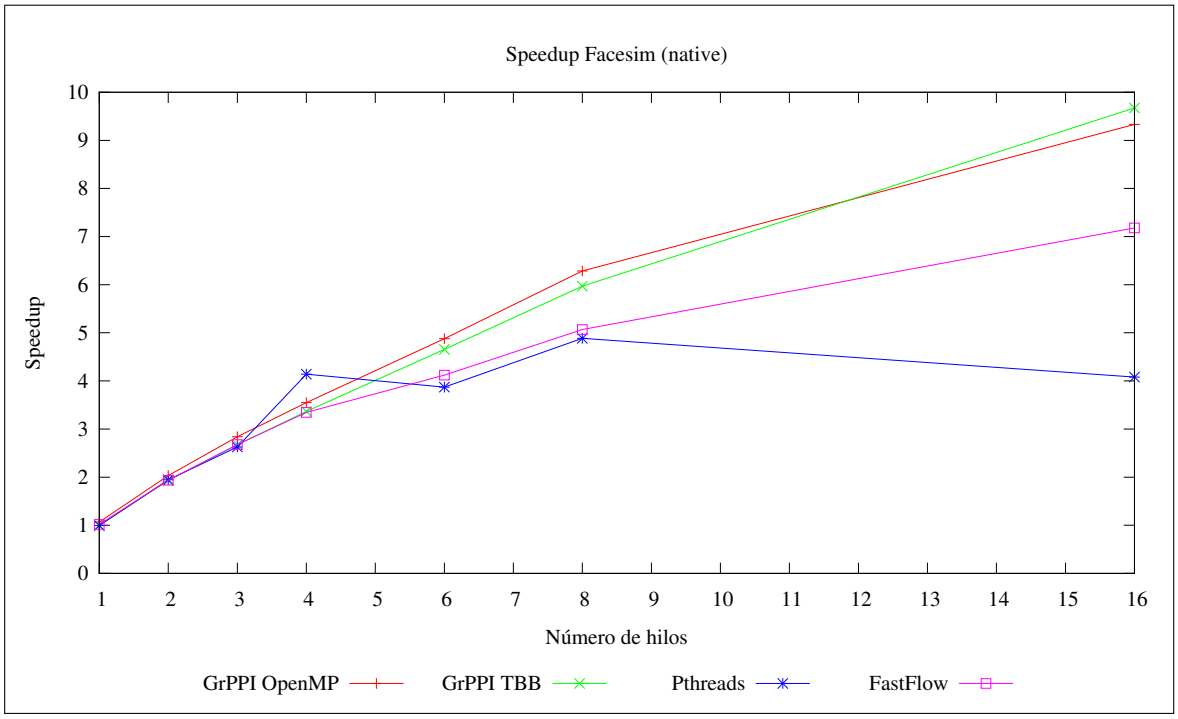


Figura 5-14: *Gráfica de los speedups para la primera versión de la implementación de la aplicación Facesim.*

Si se analiza la primera gráfica se puede ver como todas las líneas de los modelos de programación paralela tienen un comportamiento similar salvo las que corresponden con *Pthreads y GrPPI Native*, cuyos tiempos a partir de los 4 hilos comienzan a aumentar. En cuanto al resto de modelos, para identificar el mejor es necesario analizar la segunda gráfica, en la que se puede ver que los modelos con mayor escalabilidad son *GrPPI OpenMP y GrPPI TBB*.

Aunque si se quiere establecer un único modelo como el mejor se puede pasar a analizar el *speedup* medio de ambos modelos.

Modelo	Speedup Medio
GrPPI_TBB	4,180
GrPPI_OpenMP	4,283

Tabla 5.9: *Speedups medios para los modelos con mejor rendimiento para la aplicación Facesim.*

Como se puede observar en la tabla anterior, el modelo con mayor valor medio del *speedup* es *GrPPI OpenMP* de forma que dicho modelo es el más adecuado para paralelizar esta aplicación.

5.3.5. Ferret

En el caso de la aplicación *Ferret*, dependiendo del conjunto de entrada usado varían el número de imágenes de consulta que se pueden usar, el número de imágenes de la base de datos y el objetivo, es decir, el número de imágenes que se desean obtener como resultado (las X mejores). Al igual que en las aplicaciones anteriores el conjunto usado es el *native* que en este caso consta de 3.500 imágenes para realizar consultas, un base de datos compuesta por 59.695 imágenes y un objetivo de 50 imágenes.

Como se ha comentado en la Sección 4.2.5, tanto el modelo *FastFlow* como *GrPPI* tienen cuatro versiones implementadas para esta aplicación, las cuales se van a evaluar a continuación.

Versión 1: Farm

Esta versión de la implementación consta simplemente de un patrón *Farm* de forma que no es necesario realizar el estudio para determinar el valor más adecuado para el *chunksize*. A continuación, se muestra la tabla con los resultados de las medidas de dispersión para cada uno de los modelos de programación para los cuales esta aplicación tiene implementación.

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
1	serial	361089,8	609,4187395	371391,2	$\mu \in$ [360641,511 ; 361538,089]
2	fastflow	200514,6	350,312432	122718,8	$\mu \in$ [1200256,9099 ; 200772,2901]
2	pthreads	378000	0	0	$\mu \in$ [378000 ; 378000]
2	tbb	188024,4	409,3480182	167565,8	$\mu \in$ [187723,2832 ; 375747,6832]
2	*grppi_native*	207090	206347,8	422,5076331	$\mu \in$ [206658,597 ; 206037,003]
2	*grppi_tbb*	205259,8	112,2194279	12593,2	$\mu \in$ [205177,2513 ; 205342,3487]
4	fastflow	106782	179,3390643	32162,5	$\mu \in$ [106650,078 ; 106913,922]
4	pthreads	405000	0	0	$\mu \in$ [405000 ; 405000]
4	tbb	100935,2	165,1762695	27283,2	$\mu \in$ [100813,6962 ; 201748,8962]
4	*grppi_native*	112262,8	120,1632223	14439,2	$\mu \in$ [112351,1922 ; 112174,4078]
4	*grppi_tbb*	111933	29,00861941	841,5	$\mu \in$ [111911,6612 ; 111954,3388]
6	fastflow	73117,4	223,9225312	50141,3	$\mu \in$ [72952,68238 ; 73282,11762]
6	pthreads	416000	0	0	$\mu \in$ [416000 ; 416000]
6	tbb	68911,6	138,4297656	19162,8	$\mu \in$ [68809,77094 ; 137721,3709]
6	*grppi_native*	78465,6	113,0013274	12769,3	$\mu \in$ [78548,72388 ; 78382,47612]
6	*grppi_tbb*	78230,2	97,01391653	9411,7	$\mu \in$ [78158,83648 ; 78301,56352]
8	fastflow	55486	388,537643	150961,5	$\mu \in$ [55200,19136 ; 55771,80864]
8	pthreads	417000	0	0	$\mu \in$ [417000 ; 417000]
8	tbb	52262,8	272,8373142	74440,2	$\mu \in$ [52062,10062 ; 104324,9006]
8	*grppi_native*	60389,6	18,94201679	358,8	$\mu \in$ [60403,53376 ; 60375,66624]

8	*grppi_tbb*	60276,2	67,85057111	4603,7	$\mu \in$ [60226,28906 ; 60326,11094]
10	fastflow	44757,8	116,8319306	13649,7	$\mu \in$ [44671,85833 ; 44843,74167]
10	pthreads	417400	547,7225575	300000	$\mu \in$ [416997,0948 ; 417802,9052]
10	tbb	41920,6	126,1241452	15907,3	$\mu \in$ [41827,82296 ; 83748,42296]
10	*grppi_native*	49685,2	41,21528843	1698,7	$\mu \in$ [49715,518 ; 49654,882]
10	*grppi_tbb*	49488,2	83,03433025	6894,7	$\mu \in$ [49427,11987 ; 49549,28013]
12	fastflow	37613,4	556,9854576	310232,8	$\mu \in$ [37203,68099 ; 38023,11901]
12	pthreads	418400	547,7225575	300000	$\mu \in$ [417997,0948 ; 418802,9052]
12	tbb	35404,4	411,585714	169402,8	$\mu \in$ [35101,63718 ; 70506,03718]
12	*grppi_native*	42398	75,26287265	5664,5	$\mu \in$ [42453,36344 ; 42342,63656]
12	*grppi_tbb*	42456,2	84,41386142	7125,7	$\mu \in$ [42394,10509 ; 42518,29491]
14	fastflow	30052,6	259,7042549	67446,3	$\mu \in$ [29861,56132 ; 30243,63868]
14	pthreads	420000	1224,744871	1500000	$\mu \in$ [419099,0766 ; 420900,9234]
14	tbb	29980,8	51,10968597	2612,2	$\mu \in$ [29943,20367 ; 59924,00367]
14	*grppi_native*	38408,8	665,1148021	442377,7	$\mu \in$ [37919,54095 ; 38898,05905]
14	*grppi_tbb*	36118,6	520,4049385	270821,3	$\mu \in$ [36951,78255 ; 35353,01745]
16	fastflow	26720,4	316,8561188	100397,8	$\mu \in$ [26487,32035 ; 26953,47965]
16	pthreads	423200	1095,445115	1200000	$\mu \in$ [422394,1896 ; 424005,8104]
16	tbb	26572,6	256,5468768	65816,3	$\mu \in$ [26383,88389 ; 52956,48389]
16	*grppi_native*	31769,2	572,9294896	328248,2	$\mu \in$ [32190,64745 ; 31347,75255]
16	*grppi_tbb*	33643,8	610,6903471	372942,7	$\mu \in$ [33194,57563 ; 34093,02437]
18	fastflow	23795,6	212,9748811	45358,3	$\mu \in$ [25275,62947 ; 25727,17053]
18	pthreads	425600	547,7225575	300000	$\mu \in$ [23638,93548 ; 23952,26452]
18	tbb	23876,2	334,3638736	111799,2	$\mu \in$ [23630,24163 ; 47506,44163]
18	*grppi_native*	28943,2	267,6960963	71661,2	$\mu \in$ [29140,11749 ; 28746,28251]
18	*grppi_tbb*	31518,2	360,415316	129899,2	$\mu \in$ [31253,07817 ; 31783,32183]
20	fastflow	21441,8	74,32496216	5524,2	$\mu \in$ [21387,12649 ; 21496,47351]
20	pthreads	424600	547,7225575	300000	$\mu \in$ [424197,0948 ; 425002,9052]
20	tbb	7710	98,39715443	9682	$\mu \in$ [21246,66069 ; 42724,86069]
20	*grppi_native*	26438,4	51,11555536	2612,8	$\mu \in$ [26476,00065 ; 26400,79935]
20	*grppi_tbb*	29758,2	154,6502506	23916,7	$\mu \in$ [29644,43914 ; 29871,96086]
22	fastflow	19732,4	306,5726668	93986,8	$\mu \in$ [19506,88487 ; 19957,91513]
22	pthreads	425600	1140,175425	1300000	$\mu \in$ [424761,2859 ; 426438,7141]
22	tbb	19641	286,1974843	81909	$\mu \in$ [19430,47287 ; 39071,47287]
22	*grppi_native*	24567,8	86,70178776	7517,2	$\mu \in$ [24631,57791 ; 24504,02209]
22	*grppi_tbb*	28964,4	205,5122867	42235,3	$\mu \in$ [28813,22498 ; 29115,57502]

24	fastflow	18180,6	245,5815547	60310,3	$\mu \in$ [17999,94999 ; 18361,25001]
24	pthreads	425600	547,7225575	300000	$\mu \in$ [425197,0948 ; 426002,9052]
24	tbb	17992	179,5522208	32239	$\mu \in$ [17859,92122 ; 35851,92122]
24	*grppi_native*	22958,6	55,29285668	3057,3	$\mu \in$ [22999,27348 ; 22917,92652]
24	*grppi_tbb*	28480,2	108,2644909	11721,2	$\mu \in$ [28400,56054 ; 28559,83946]

Tabla 5.10: *Medidas de dispersión de los tiempos de ejecución obtenidos para la versión Pipeline de Farms de la aplicación Ferret.*

Las medias de los tiempos de ejecución para cada uno de los modelos de programación paralela que se muestran en la tabla anterior se han representado también en la siguiente gráfica para que se pudiera determinar de forma más sencilla el modelo más eficiente para esta versión de la aplicación.

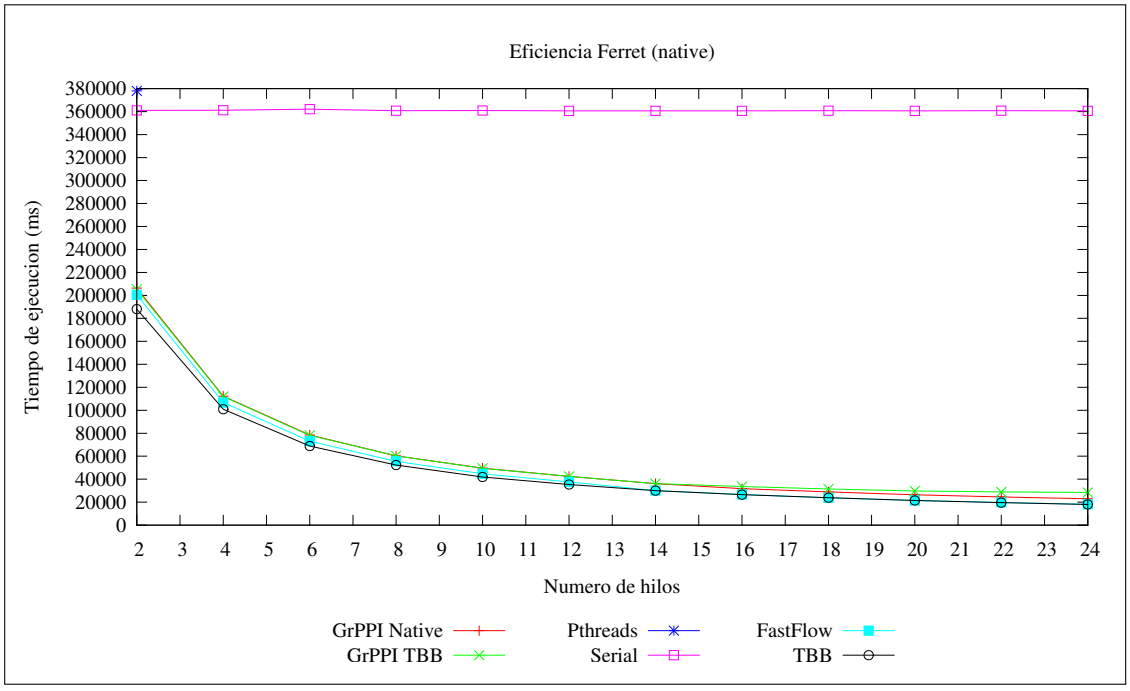


Figura 5-15: *Gráfica de tiempos de ejecución medios para la primera versión de la implementación de la aplicación Ferret.*

A parte del tiempo de ejecución medio, también se podría considerar otra medida para la evaluación de los modelos, esta medida es el *speedup* o aumento de la eficiencia con respecto a la versión serial, lo cual se puede observar en la siguiente gráfica:

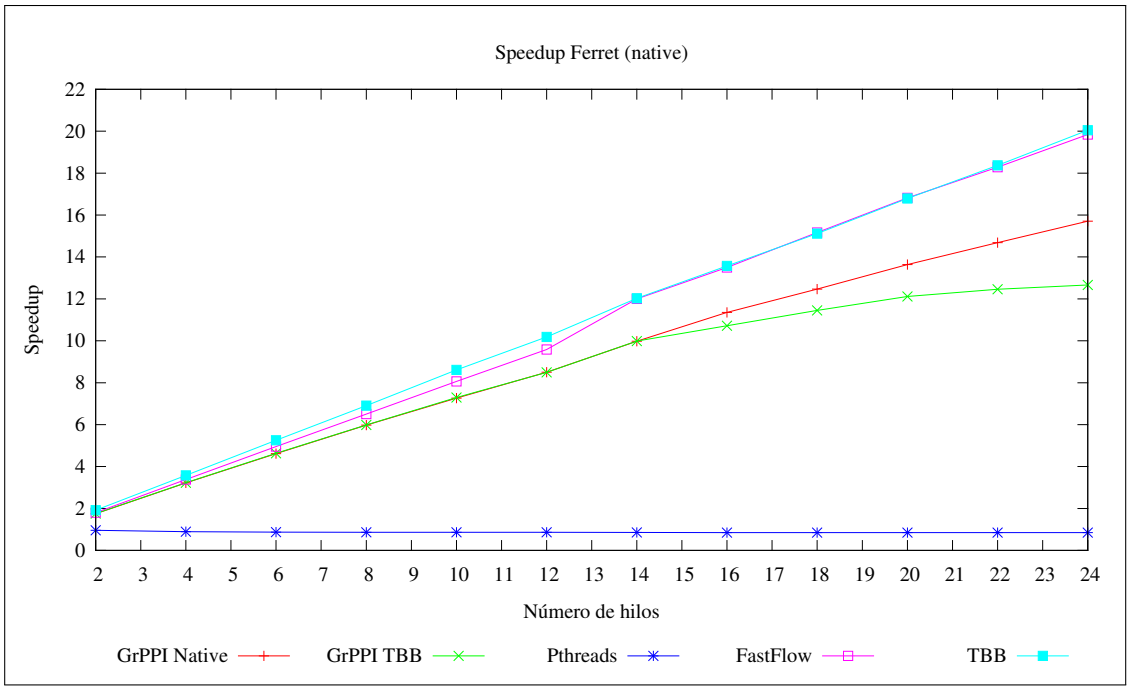


Figura 5-16: *Gráfica de los speedups para la primera versión de la implementación de la aplicación Ferret.*

Al analizar la primera gráfica, se puede observar que todos los modelos de programación paralela tienen líneas de progresión muy similares, por lo tanto, si se quieren estudiar las diferencias entre los modelos es necesario consultar la segunda gráfica. En ésta se puede ver como los modelos con mayor escalabilidad son *FastFlow y TBB* seguidos por *GrPPI Native y GrPPI TBB*, aunque si se tuviera que escoger un único modelo como el mejor sería *TBB* debido a la ligera bajada en el *speedup* que experimenta el modelo *FastFlow* en el rango entre 6 y 14 del *eje x*.

Versión 2: Farm Optimizado

Esta segunda versión es similar a la anterior y al seguir usando un patrón de tipo *Farm* se evita tener que realizar el estudio del *chunksize*. En la siguiente tabla se muestran las medidas de dispersión asociadas a los dos modelos de programación que disponen de esta implementación, *FastFlow y GrPPI*.

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
2	fastflow	200648	471,009554	221850	$\mu \in$ [200301,5249 ; 200994,4751]
2	*grppi_native*	204021,8	212,5410549	45173,7	$\mu \in$ [203865,4546 ; 204178,1454]
2	*grppi_tbb*	203451,4	317,2558904	100651,3	$\mu \in$ [203218,0263 ; 203684,7737]
4	fastflow	106578,8	198,7151227	39487,7	$\mu \in$ [106432,625 ; 106724,975]
4	*grppi_native*	108729	157,3324506	24753,5	$\mu \in$ [108613,2661 ; 108844,7339]
4	*grppi_tbb*	108533,8	183,1589474	33547,2	$\mu \in$ [108399,0681 ; 108668,5319]

6	fastflow	73018,4	97,74354199	9553,8	$\mu \in [72946{,}49976 ; 73090{,}30024]$
6	*grppi_native*	74308	52,04325124	2708,5	$\mu \in [74269{,}71694 ; 74346{,}28306]$
6	*grppi_tbb*	74151,6	49,24733495	2425,3	$\mu \in [74115{,}37362 ; 74187{,}82638]$
8	fastflow	55168,2	116,0568826	13469,2	$\mu \in [55082{,}82845 ; 55253{,}57155]$
8	*grppi_native*	56185,2	197,9108385	39168,7	$\mu \in [56039{,}61661 ; 56330{,}78339]$
8	*grppi_tbb*	56386,4	421,889559	177990,8	$\mu \in [56076{,}05766 ; 56696{,}74234]$
10	fastflow	44507	83,0572092	6898,5	$\mu \in [44445{,}90304 ; 44568{,}09696]$
10	*grppi_native*	45213,6	151,3234945	22898,8	$\mu \in [45102{,}2863 ; 45324{,}9137]$
10	*grppi_tbb*	45268	429,2540041	184259	$\mu \in [44952{,}24037 ; 45583{,}75963]$
12	fastflow	37539,4	279,9085565	78348,8	$\mu \in [37333{,}49901 ; 37745{,}30099]$
12	*grppi_native*	37866,6	42,16396566	1777,8	$\mu \in [37835{,}58415 ; 37897{,}61585]$
12	*grppi_tbb*	37873,8	42,76330202	1828,7	$\mu \in [37842{,}34328 ; 37905{,}25672]$
14	fastflow	29909,2	166,9002696	27855,7	$\mu \in [29786{,}42801 ; 30031{,}97199]$
14	*grppi_native*	30359,2	40,86808045	1670,2	$\mu \in [30329{,}1374 ; 30389{,}2626]$
14	*grppi_tbb*	30877	337,9230682	114192	$\mu \in [30628{,}42349 ; 31125{,}57651]$
16	fastflow	26796,4	471,0544555	221892,3	$\mu \in [26449{,}89192 ; 27142{,}90808]$
16	*grppi_native*	26951,2	253,7197273	64373,7	$\mu \in [26764{,}56354 ; 27137{,}83646]$
16	*grppi_tbb*	28105,8	394,6209574	155725,7	$\mu \in [27815{,}51647 ; 28396{,}08353]$
18	fastflow	23542,8	52,04997598	2709,2	$\mu \in [23504{,}51199 ; 23581{,}08801]$
18	*grppi_native*	24171,8	341,2985497	116484,7	$\mu \in [23920{,}74048 ; 24422{,}85952]$
18	*grppi_tbb*	26056,4	425,5059342	181055,3	$\mu \in [25743{,}39745 ; 26369{,}40255]$
20	fastflow	21445,4	172,5479643	29772,8	$\mu \in [21318{,}47356 ; 21572{,}32644]$
20	*grppi_native*	21650,2	78,88726133	6223,2	$\mu \in [19690{,}03593 ; 19707{,}96407]$
20	*grppi_tbb*	24472,8	183,38266	33629,2	$\mu \in [24337{,}90355 ; 24607{,}69645]$
22	fastflow	19852	336,59694	113297,5	$\mu \in [19604{,}39899 ; 20099{,}60101]$
22	*grppi_native*	19699	12,18605761	148,5	$\mu \in [24387{,}86726 ; 24462{,}53274]$
22	*grppi_tbb*	23370,4	36,30840123	1318,3	$\mu \in [23343{,}69151 ; 23397{,}10849]$
24	fastflow	17927,8	139,8470593	19557,2	$\mu \in [17824{,}92838 ; 18030{,}67162]$
24	*grppi_native*	18139,8	8,228000972	67,7	$\mu \in [18133{,}74748 ; 18145{,}85252]$
24	*grppi_tbb*	22619	176,1973326	31045,5	$\mu \in [22489{,}38908 ; 22748{,}61092]$

Tabla 5.11: *Medidas de dispersión de los tiempos de ejecución obtenidos para la versión Farm Optimizado de la aplicación Ferret.*

A continuación, se muestran las medias de la tabla anterior en conjunto con las medias que son comunes para las cuatro versiones (*Serial, Pthreads y TBB*) en forma de gráfica.

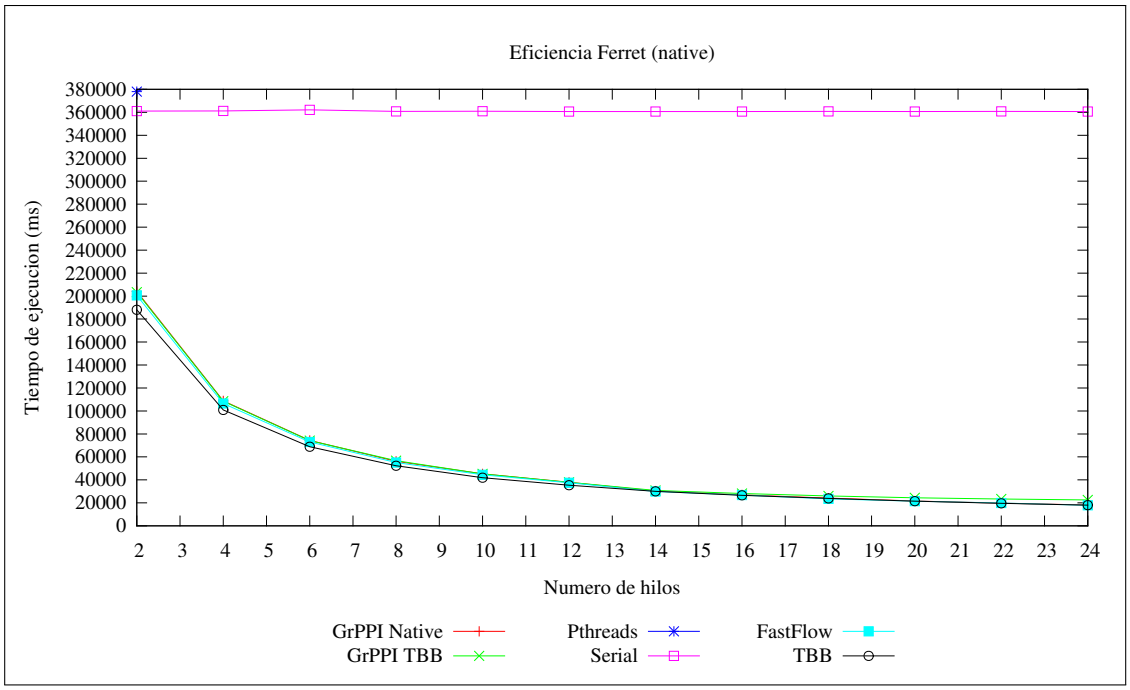


Figura 5-17: *Gráfica de tiempos de ejecución medios para la segunda versión de la implementación de la aplicación Ferret.*

A parte del tiempo de ejecución medio, también se podría considerar otra medida para la evaluación de los modelos, esta medida es el *speedup* o aumento de la eficiencia con respecto a la versión serial, lo cual se puede observar en la siguiente gráfica:

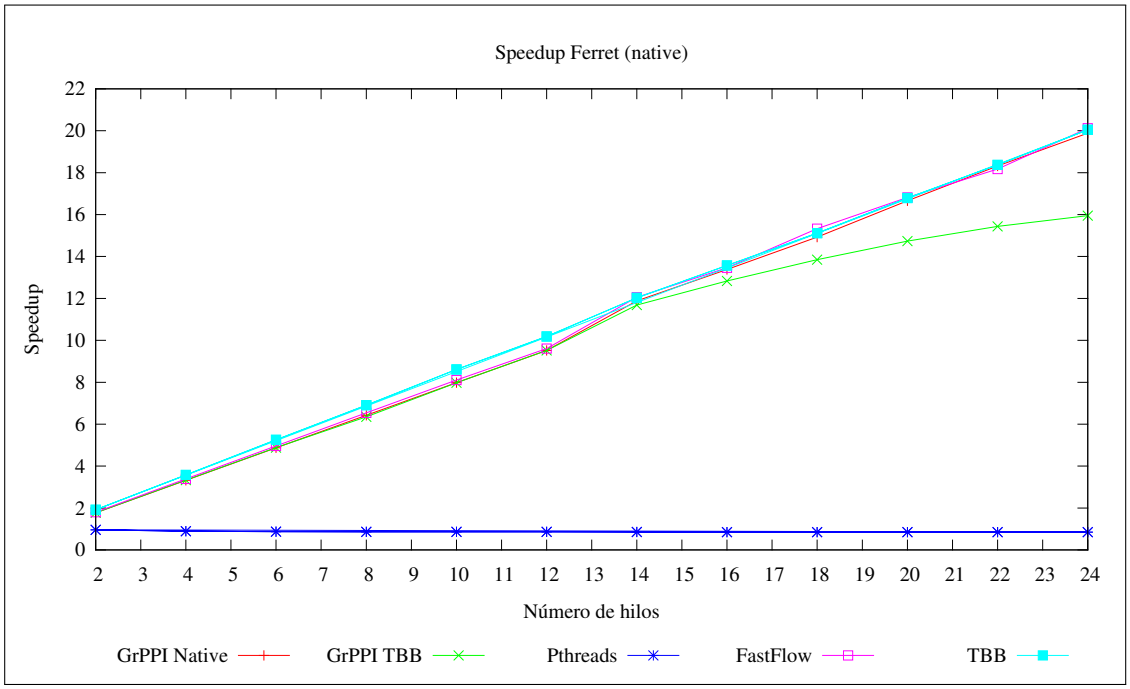


Figura 5-18: *Gráfica de los speedups para la segunda versión de la implementación de la aplicación Ferret.*

Si se analiza la primera gráfica se puede ver que todas las líneas excepto la serial coinciden, de forma que para establecer cual o cuales son los mejores modelos para esta versión hay que fijarse en la segunda gráfica. En dicha gráfica se puede observar como las líneas de *GrPPI Native, GrPPI TBB y FastFlow* están ligeramente por debajo de la línea de *TBB* en el rango de 0 a 14 hilos, mientras que en la segunda parte de la gráfica (rango de 14 a 24 hilos) la línea correspondiente al modelo *GrPPI TBB* se aparta del resto y las líneas de *GrPPI Native y FatFlow* coinciden con la línea de *TBB*. Teniendo en cuenta todos estos aspectos se puede concluir que el mejor modelo para esta versión sería *TBB* ya que su progresión es estable y superior a la del resto de modelos.

Versión 3: *Pipeline* de *Farms*

Esta implementación consta de un patrón *Pipeline* en el que sus etapas intermedias son patrones *Farm* permitiendo su ejecución en paralelo. Al no usar el patrón *Parallel_for*, no es necesario hacer un estudio del *chunksize*, de forma que a continuación se muestran directamente los resultados de las medidas de dispersión de los modelos para los cuales se ha cambiado la implementación (*GrPPI* y *FastFlow*).

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
2	fastflow	191452,2	398,7689807	159016,7	$\mu \in$ [191158,8652 ; 191745,5348]
2	*grppi_native*	187802,8	228,9829688	52433,2	$\mu \in$ [187634,3599 ; 187971,2401]
2	*grppi_tbb*	187528,2	353,9656763	125291,7	$\mu \in$ [187267,8225 ; 187788,5775]
4	fastflow	102028,2	201,1683375	40468,7	$\mu \in$ [101880,2204 ; 102176,1796]
4	*grppi_native*	103025,2	165,8604835	27509,7	$\mu \in$ [102903,1929 ; 103147,2071]
4	*grppi_tbb*	102729,2	26,26214005	689,7	$\mu \in$ [102709,8815 ; 102748,5185]
6	fastflow	70267,2	120,7091546	14570,7	$\mu \in$ [70178,40624 ; 70355,99376]
6	*grppi_native*	72246	67,63505008	4574,5	$\mu \in$ [72196,2476 ; 72295,7524]
6	*grppi_tbb*	71947,6	45,92711617	2109,3	$\mu \in$ [71913,81597 ; 71981,38403]
8	fastflow	52927,2	186,0274173	34606,2	$\mu \in$ [52790,35806 ; 53064,04194]
8	*grppi_native*	55748,2	72,74407192	5291,7	$\mu \in$ [55694,68939 ; 55801,71061]
8	*grppi_tbb*	55485,4	33,5454915	1125,3	$\mu \in$ [55460,72391 ; 55510,07609]
10	fastflow	42650,8	64,79737649	4198,7	$\mu \in$ [42603,13499 ; 42698,46501]
10	*grppi_native*	45911,4	166,958977	27875,3	$\mu \in$ [45788,58483 ; 46034,21517]
10	*grppi_tbb*	45520,4	96,58312482	9328,3	$\mu \in$ [45449,35337 ; 45591,44663]
12	fastflow	35702,8	114,9986956	13224,7	$\mu \in$ [35618,20686 ; 35787,39314]
12	*grppi_native*	39166,8	48,6538796	2367,2	$\mu \in$ [39131,01016 ; 39202,58984]
12	*grppi_tbb*	39130,2	56,94471003	3242,7	$\mu \in$ [39088,31142 ; 39172,08858]

14	fastflow	35895,8	2446,691991	5986301,7	$\mu \in [34096{,}01116 ; 37695{,}58884]$
14	*grppi_native*	38408,8	665,1148021	442377,7	$\mu \in [37919{,}54095 ; 38898{,}05905]$
14	*grppi_tbb*	35344,2	408,0082107	166470,7	$\mu \in [35044{,}06879 ; 35644{,}33121]$
16	fastflow	30115,4	569,9537701	324847,3	$\mu \in [29696{,}14149 ; 30534{,}65851]$
16	*grppi_native*	32752,8	169,3788653	28689,2	$\mu \in [32628{,}20475 ; 32877{,}39525]$
16	*grppi_tbb*	32724,6	780,2504726	608790,8	$\mu \in [32150{,}64705 ; 33298{,}55295]$
18	fastflow	25501,4	306,9198592	94199,8	$\mu \in [25275{,}62947 ; 25727{,}17053]$
18	*grppi_native*	29035	132,1041256	17451,5	$\mu \in [28937{,}82409 ; 29132{,}17591]$
18	*grppi_tbb*	30913,8	164,2549847	26979,7	$\mu \in [30792{,}97388 ; 31034{,}62612]$
20	fastflow	22431,4	100,2162661	10043,3	$\mu \in [22357{,}68082 ; 22505{,}11918]$
20	*grppi_native*	26539,4	84,51508741	7142,8	$\mu \in [26477{,}23063 ; 26601{,}56937]$
20	*grppi_tbb*	29405,6	410,5926205	168586,3	$\mu \in [29103{,}5677 ; 29707{,}6323]$
22	fastflow	20261	111,1912766	12363,5	$\mu \in [20179{,}2076 ; 20342{,}7924]$
22	*grppi_native*	24425,2	50,75135466	2575,7	$\mu \in [24387{,}86726 ; 24462{,}53274]$
22	*grppi_tbb*	28133,8	106,6287954	11369,7	$\mu \in [28055{,}36376 ; 28212{,}23624]$
24	fastflow	18652	162,7774554	26496,5	$\mu \in [18532{,}26076 ; 18771{,}73924]$
24	*grppi_native*	22902,8	56,63214635	3207,2	$\mu \in [22861{,}14134 ; 22944{,}45866]$
24	*grppi_tbb*	27527,2	157,0277046	24657,7	$\mu \in [27411{,}69028 ; 27642{,}70972]$

Tabla 5.12: *Medidas de dispersión de los tiempos de ejecución obtenidos para la versión Pipeline de Farms de la aplicación Ferret.*

Para una mejor evaluación, los resultados de la tabla se han representado en forma de gráfica:

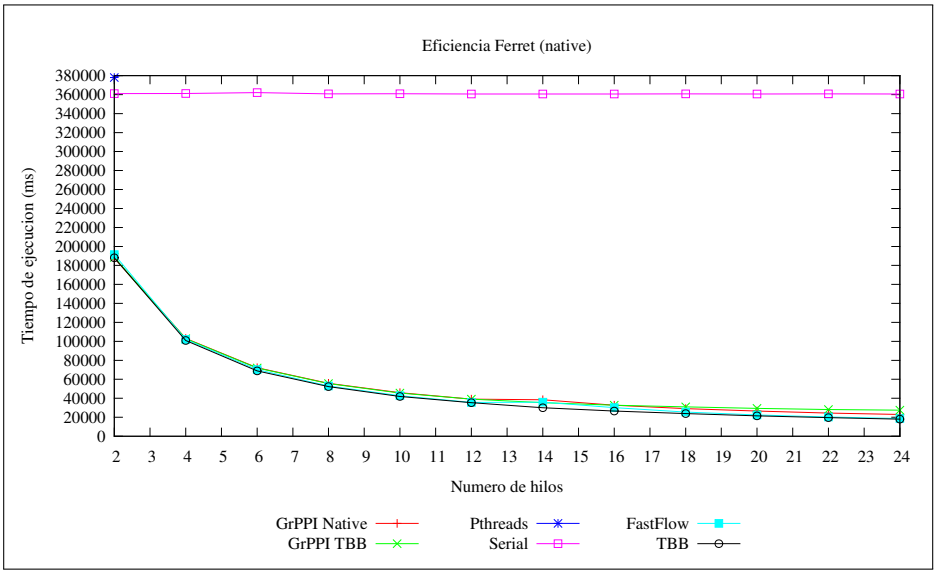


Figura 5-19: *Gráfica de tiempos de ejecución medios para los distintos modelos de programación para la versión 3 de la aplicación Ferret.*

Otra medida que se puede estudiar es el *speedup* o aumento de la eficiencia con respecto al modelo serial, lo cual se puede observar en la siguiente gráfica:

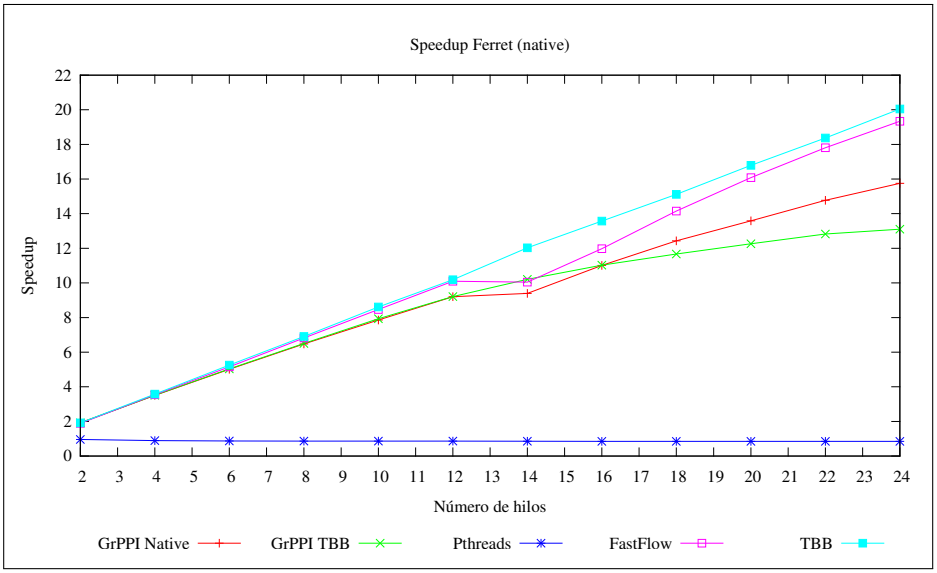


Figura 5-20: *Gráfica de los speedups para los distintos modelos de programación de la aplicación Ferret.*

Como se puede ver en la primera gráfica, la línea para el modelo de programación *Pthreads* ni siquiera aparece en la gráfica, esto es debido a que va peor que la versión secuencial, lo que se puede apreciar también en la gráfica de *speedups* donde la línea está por debajo del 1.

En cuanto al resto de modelos, si se analiza simplemente la primera gráfica no se puede distinguir ninguno que vaya claramente mejor que el resto, por lo que se debe pasar a estudiar la segunda gráfica. En el rango de 0 a 6 hilos de ésta se puede observar como todos los modelos salvo *Phtreads* van a la par, sin embargo, a partir de los 6 hilos se empiezan a descolgar las líneas correspondientes con los modelos *GrPPI TBB y GrPPI Native*. Y más adelante, a partir de 12 hilos el *speedup* de *FastFlow* empieza a disminuir, lo que significa que el mejor modelo de este experimento sería *TBB* pues su línea se mantiene en todo momento por encima del resto.

Versión 4: Farm de Pipelines

Esta última versión contiene un patrón *Farm* y en su interior un patrón *Pipeline*, por lo que no es necesario realizar el estudio del *chunksize*. En este caso simplemente se van a mostrar en la tabla inferior las medidas de dispersión correspondientes con los modelos para los cuales varía la implementación (*GrPPI* y *FastFlow*).

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
2	fastflow	203623,2	3518,489122	12379765,7	$\mu \in$ [201034,9962 ; 206211,4038]
2	*grppi_native*	205104	1378,05624	1899039	$\mu \in$ [204090,3006 ; 206117,6994]
2	*grppi_tbb*	202757,4	285,3196453	81407,3	$\mu \in$ [202547,5186 ; 202967,2814]

4	fastflow	113747,6	3489,16993	12174306,8	$\mu \in$ [111180,9634 ; 116314,2366]
4	*grppi_native*	113031,8	957,0552231	915954,7	$\mu \in$ [112327,7893 ; 113735,8107]
4	*grppi_tbb*	110688,8	229,0779343	52476,7	$\mu \in$ [110520,2901 ; 110857,3099]
6	fastflow	78546,8	1679,35172	2820222,2	$\mu \in$ [77311,46736 ; 79782,13264]
6	*grppi_native*	80596,6	1097,992168	1205586,8	$\mu \in$ [79788,91597 ; 81404,28403]
6	*grppi_tbb*	77194,4	301,5954244	90959,8	$\mu \in$ [76972,54613 ; 77416,25387]
8	fastflow	59642	293,8749734	86362,5	$\mu \in$ [59425,8253 ; 59858,1747]
8	*grppi_native*	62481,6	507,8831559	257945,3	$\mu \in$ [62108,00069 ; 62855,19931]
8	*grppi_tbb*	59430,8	62,60351428	3919,2	$\mu \in$ [59384,7488 ; 59476,8512]
10	fastflow	47145,4	279,8281616	78303,8	$\mu \in$ [46939,55815 ; 47351,24185]
10	*grppi_native*	52489,2	1165,55253	1358512,7	$\mu \in$ [51631,8185 ; 53346,5815]
10	*grppi_tbb*	48836,8	115,2852983	13290,7	$\mu \in$ [48751,99603 ; 48921,60397]
12	fastflow	41393,2	316,2929022	100041,2	$\mu \in$ [41160,53466 ; 41625,86534]
12	*grppi_native*	45103,2	1219,943933	1488263,2	$\mu \in$ [44205,80814 ; 46000,59186]
12	*grppi_tbb*	41807,8	50,03698632	2503,7	$\mu \in$ [41770,99275 ; 41844,60725]
14	fastflow	46260,6	798,143972	637033,8	$\mu \in$ [45673,48457 ; 46847,71543]
14	*grppi_native*	48164,6	1973,272485	3893804,3	$\mu \in$ [46713,05898 ; 49616,14102]
14	*grppi_tbb*	35745	673,4571256	453544,5	$\mu \in$ [35249,60433 ; 36240,39567]
16	fastflow	34643	408,9345913	167227,5	$\mu \in$ [34342,18734 ; 34943,81266]
16	*grppi_native*	40674,2	2650,659673	7025996,7	$\mu \in$ [38724,37235 ; 42624,02765]
16	*grppi_tbb*	32622	627,8984791	394256,5	$\mu \in$ [32160,11731 ; 33083,88269]
18	fastflow	30082,6	1242,4139	1543592,3	$\mu \in$ [29168,67921 ; 30996,52079]
18	*grppi_native*	37700	1330,204871	1769445	$\mu \in$ [36721,50009 ; 38678,49991]
18	*grppi_tbb*	31068,6	292,7495517	85702,3	$\mu \in$ [30853,25316 ; 31283,94684]
20	fastflow	24705,8	693,4145225	480823,7	$\mu \in$ [24195,72365 ; 25215,87635]
20	*grppi_native*	31999	1871,745041	3503429,5	$\mu \in$ [30622,14265 ; 33375,85735]
20	*grppi_tbb*	29215,4	426,126507	181583,8	$\mu \in$ [28901,94096 ; 29528,85904]
22	fastflow	20781,4	236,5212464	55942,3	$\mu \in$ [20607,41476 ; 20955,38524]
22	*grppi_native*	29254	2187,4812	4785074	$\mu \in$ [27644,88685 ; 30863,11315]
22	*grppi_tbb*	27956,8	208,6305347	43526,7	$\mu \in$ [27803,33119 ; 28110,26881]
24	fastflow	19635	83,48353131	6969,5	$\mu \in$ [19573,58944 ; 19696,41056]
24	*grppi_native*	26530,4	557,1734918	310442,3	$\mu \in$ [26120,54268 ; 26940,25732]
24	*grppi_tbb*	27326,6	78,34730372	6138,3	$\mu \in$ [27268,96765 ; 27384,23235]

Tabla 5.13: *Medidas de dispersión de los tiempos de ejecución obtenidos para la versión Farm de Pipelines de la aplicación Ferret.*

Con el fin de tener una idea más clara de los datos mostrados en la tabla anterior, se ha decidido representar las medias obtenidas en una gráfica.

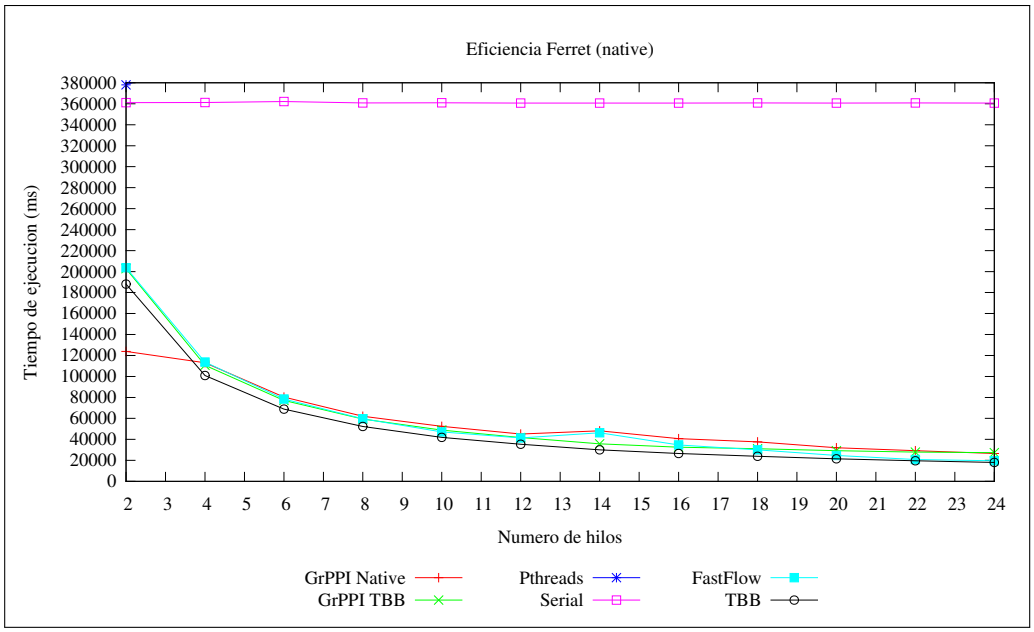


Figura 5-21: *Gráfica de tiempos de ejecución medios para la última versión de la implementación de la aplicación Ferret.*

A parte del tiempo de ejecución medio, también se podría considerar otra medida para la evaluación de los modelos, esta medida es el *speedup* o aumento de la eficiencia con respecto a la versión serial, lo cual se puede observar en la siguiente gráfica:

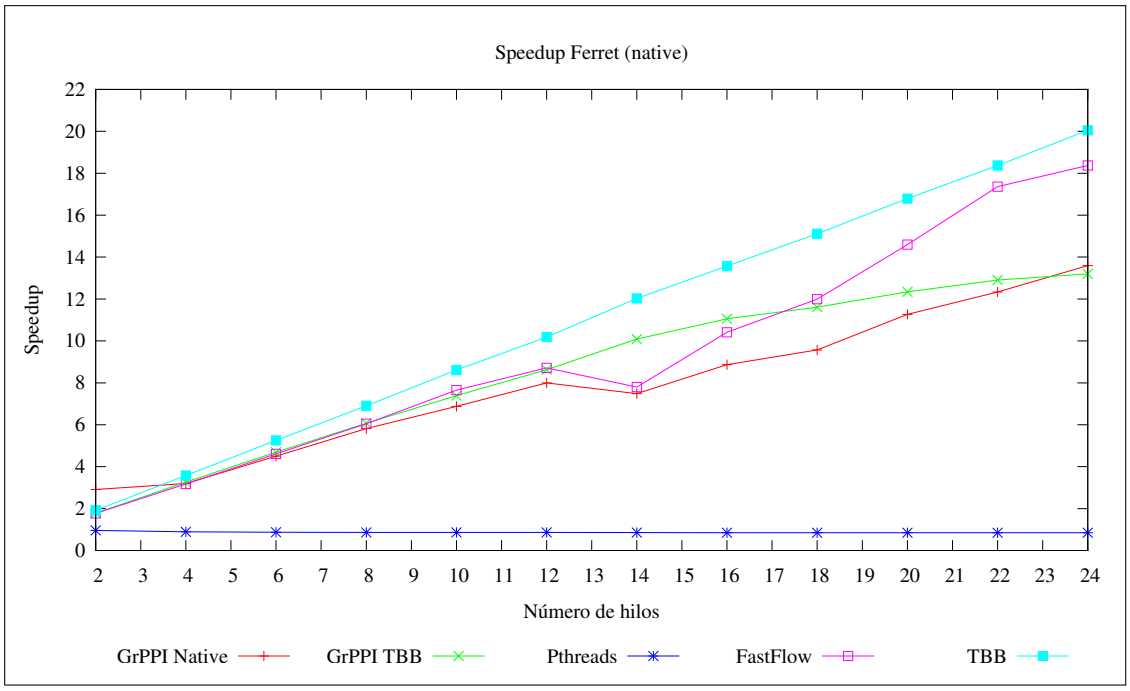


Figura 5-22: *Gráfica de los speedups para la última versión de la implementación de la aplicación Ferret.*

Al analizar la primera gráfica expuesta no se ve claro que modelo de programación es el más eficiente,

sin embargo, si se observa la segunda gráfica se puede distinguir claramente que el modelo con mayor escalabilidad es *TBB*.

Si se analizaran las cuatro versiones expuestas, se podría concluir que la mejor de ellas es la segunda versión (*Farm Optimizado*) ya que es la versión en la que tanto el modelo *GrPPI* como *FastFlow* siguen una progresión más cercana al modelo escogido como el mejor para esta aplicación (*TBB*).

5.3.6. Fluidanimate

En el caso de la aplicación *Fluidanimate*, dependiendo del conjunto de entrada cambia el número de partículas que se usan y el número de *frames*. En concreto para el conjunto *native* se usan 500.000 partículas y un total de 500 *frames*.

Esta aplicación hace uso del patrón *Parallel_for* de *GrPPI* de forma que se ha tenido que realizar un análisis del valor del *chunksize*. Para ello se ha ejecutado el programa varias veces usando 2, 4, 8 y 16 hilos puesto que este programa solo acepta números de hilos potencias de 2, y se ha ido cambiando el valor del *chunksize* de 1 en 1 en un rango entre 1 y el número de hilos de dicha ejecución. Los resultados de dichas ejecuciones se pueden ver en la siguiente gráfica:

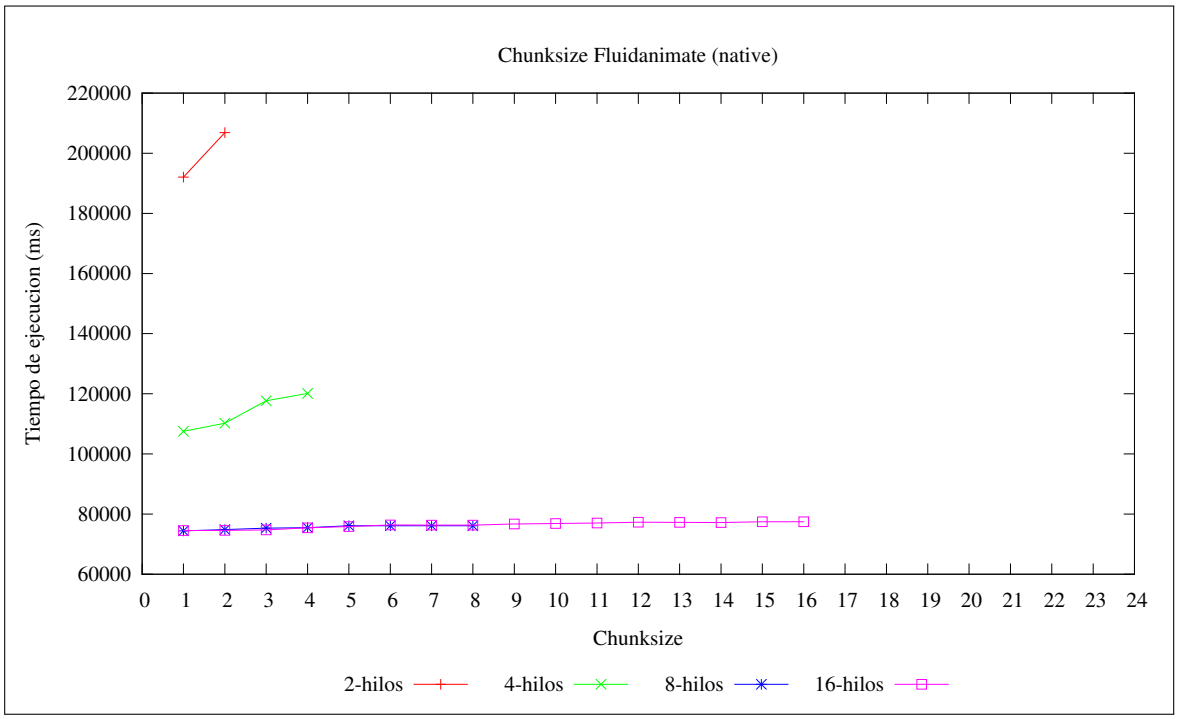


Figura 5-23: *Gráfica de tiempos de ejecución obtenidos mediante la versión GrPPI Native de la aplicación Fluidanimate para los distintos valores del chunksize.*

Como se puede observar en la gráfica anterior, independientemente del valor del *chunksize* el tiempo de

ejecución es estable en las versiones de 8 y 16 hilos, por lo que para elegir el valor hay que fijarse en las líneas para 2 y 4 hilos que tienen su mínimo con el valor 1.

A continuación, se muestra la tabla con las medidas de dispersión calculadas a partir de los tiempos de ejecución obtenidos para los distintos modelos de programación.

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
1	serial	322177	310,7796647	96584	$\mu \in$ [321948,3902 ; 322405,6098]
2	fastflow	189061,2	254,1538904	64594,2	$\mu \in$ [188874,2442 ; 189248,1558]
2	pthreads	186941	283,1501369	80174	$\mu \in$ [186732,7145 ; 187149,2855]
2	*grppi_native*	202286,4	18733,40306	350940390,3	$\mu \in$ [188506,0918 ; 216066,7082]
2	*grppi_openmp*	188493,6	72,94724121	5321,3	$\mu \in$ [188439,9399 ; 188547,2601]
2	*grppi_tbb*	191249,8	51,10968597	2612,2	$\mu \in$ [191212,2037 ; 191287,3963]
4	fastflow	105588,2	171,7009027	29481,2	$\mu \in$ [105461,8967 ; 105714,5033]
4	pthreads	103851,6	255,5216234	65291,3	$\mu \in$ [103663,6381 ; 104039,5619]
4	*grppi_native*	106661,8	478,4362026	228901,2	$\mu \in$ [106309,8619 ; 107013,7381]
4	*grppi_openmp*	103669,6	90,84217082	8252,3	$\mu \in$ [103602,7764 ; 103736,4236]
4	*grppi_tbb*	105295,6	131,188033	17210,3	$\mu \in$ [105199,098 ; 105392,102]
8	fastflow	68188	83,23761169	6928,5	$\mu \in$ [68126,77034 ; 68249,22966]
8	pthreads	67676,8	145,0334444	21034,7	$\mu \in$ [67570,11327 ; 67783,48673]
8	*grppi_native*	74773,4	427,2491077	182541,8	$\mu \in$ [74459,11517 ; 75087,68483]
8	*grppi_openmp*	59528,4	91,30607866	8336,8	$\mu \in$ [59461,23517 ; 59595,56483]
8	*grppi_tbb*	68327,8	69,94783771	4892,7	$\mu \in$ [68276,34631 ; 68379,25369]
16	fastflow	39717,6	352,9409299	124567,3	$\mu \in$ [39457,97633 ; 39977,22367]
16	pthreads	40957,8	281,9178958	79477,7	$\mu \in$ [40750,42094 ; 41165,17906]
16	*grppi_native*	76697,4	1357,785071	1843580,3	$\mu \in$ [75698,61207 ; 77696,18793]
16	*grppi_openmp*	36744	1827,402118	3339398,5	$\mu \in$ [35399,76135 ; 38088,23865]
16	*grppi_tbb*	42139,8	4576,719043	20946357,2	$\mu \in$ [38773,16133 ; 45506,43867]

Tabla 5.14: *Medidas de dispersión de los tiempos de ejecución obtenidos para la aplicación Fluidanimate.*

En la siguiente gráfica se pueden ver representadas las medias que se muestran en la tabla anterior de forma que se pueda evaluar mejor cual sería el mejor modelo de programación para esta aplicación concreta.

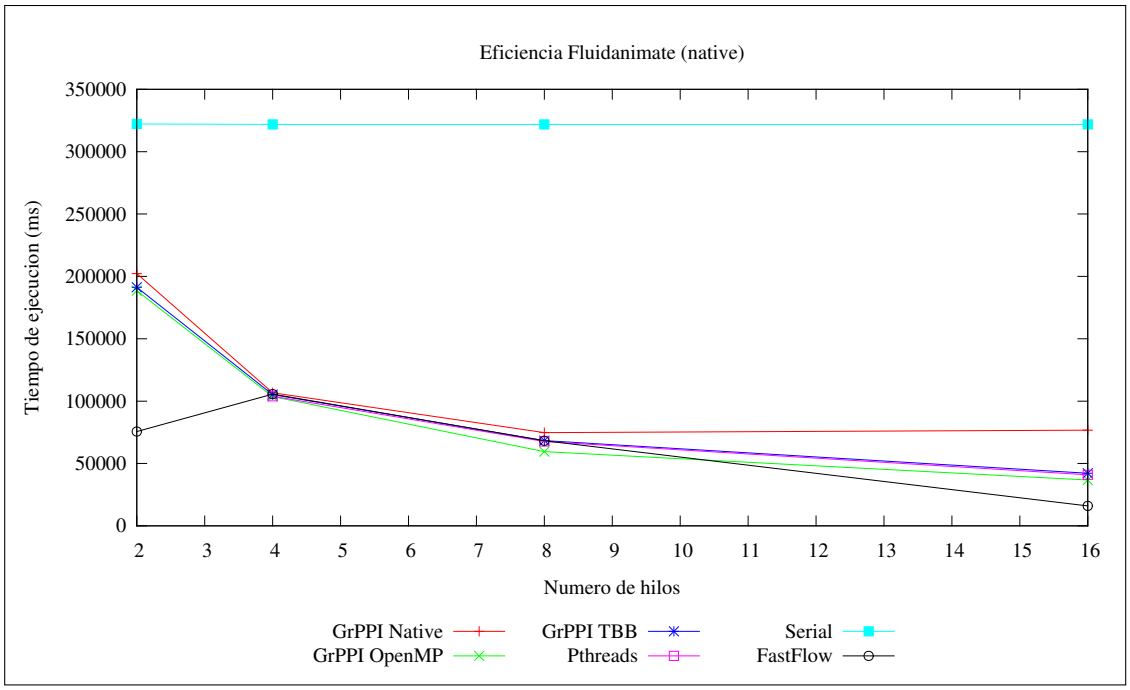


Figura 5-24: *Gráfica de tiempos de ejecución medios para los distintos modelos de programación de la aplicación Fluidanimate.*

Otra medida que se puede estudiar es el *speedup* o aumento de la eficiencia con respecto al modelo serial, lo cual se puede observar en la siguiente gráfica:

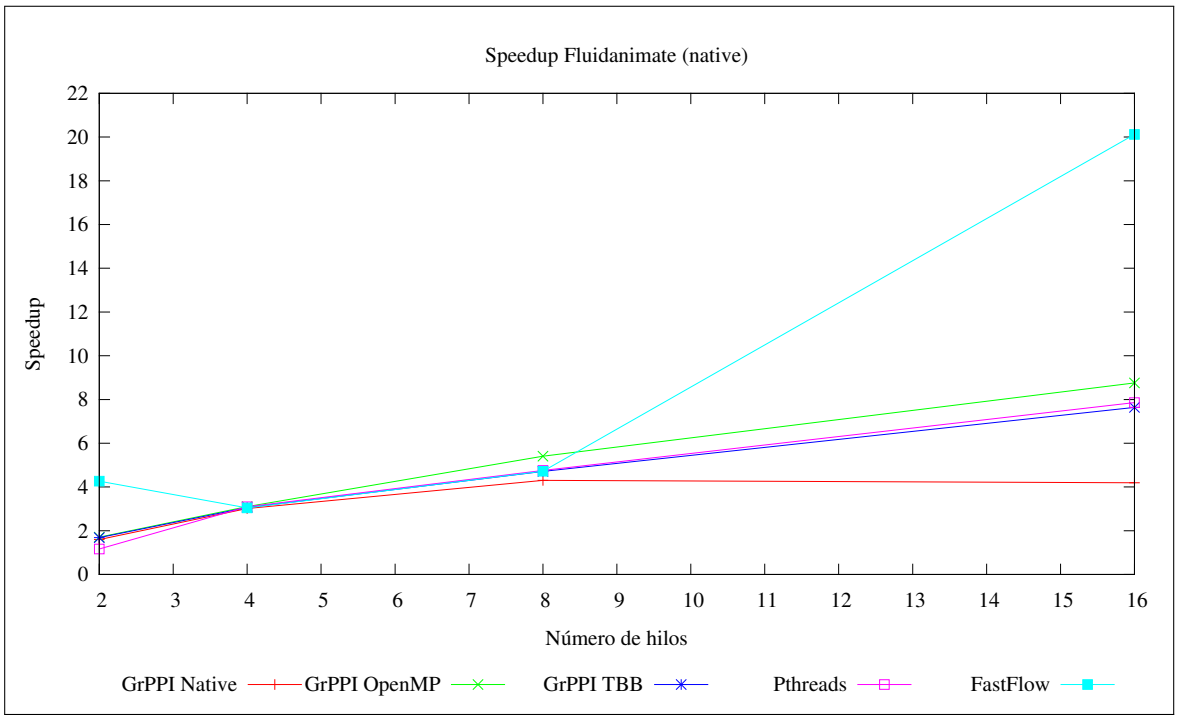


Figura 5-25: *Gráfica de los speedups para los distintos modelos de programación de la aplicación Fluidanimate.*

Como se puede ver si se evalúa la primera de las dos gráficas, las medias de los tiempos de ejecución para los distintos modelos están bastante igualadas, salvo casos como el de *FastFlow* con 2 hilos y la ejecución con 16 hilos en la que los modelos se separan más.

Sin embargo, al analizar la segunda gráfica, se puede ver claramente que el modelo con mayor escalabilidad es *FastFlow* a pesar de que presenta un comportamiento extraño puesto que empieza con un tiempo de ejecución para la versión de 2 hilos muy bajo en comparación con el resto, pero en la ejecución con 4 hilos dicho tiempo aumenta y luego vuelve a disminuir con 8 y 16 hilos.

5.3.7. Raytrace

En el caso de la aplicación *Raytrace*, dependiendo del conjunto de entrada cambia la resolución en píxeles, el número de polígonos, el objeto de entrada y el número de *frames*. En el caso del conjunto *native* se establece una resolución de 1920 x 1080 píxeles, 10 millones de polígonos, un total de 200 *frames* y el objeto usado en el análisis es *"Thai Statue"*.

En primer lugar, se ha realizado un análisis para determinar cuál es el valor más adecuado para el *chunksize*, valor necesario para poder hacer las divisiones para la ejecución del programa con la versión de *Native* de *GrPPI* cuando se emplea el patrón *Parallel_for*. Para ello se ha ejecutado el programa usando 6, 12, 18 y 24 hilos, variando en cada ejecución en 50 unidades el valor del *chunksize* entre los valores 1 y 2000, para determinar el punto mínimo del que el tiempo de ejecución en todos los casos.

A continuación, se puede ver una gráfica de los resultados obtenidos de dichas ejecuciones:

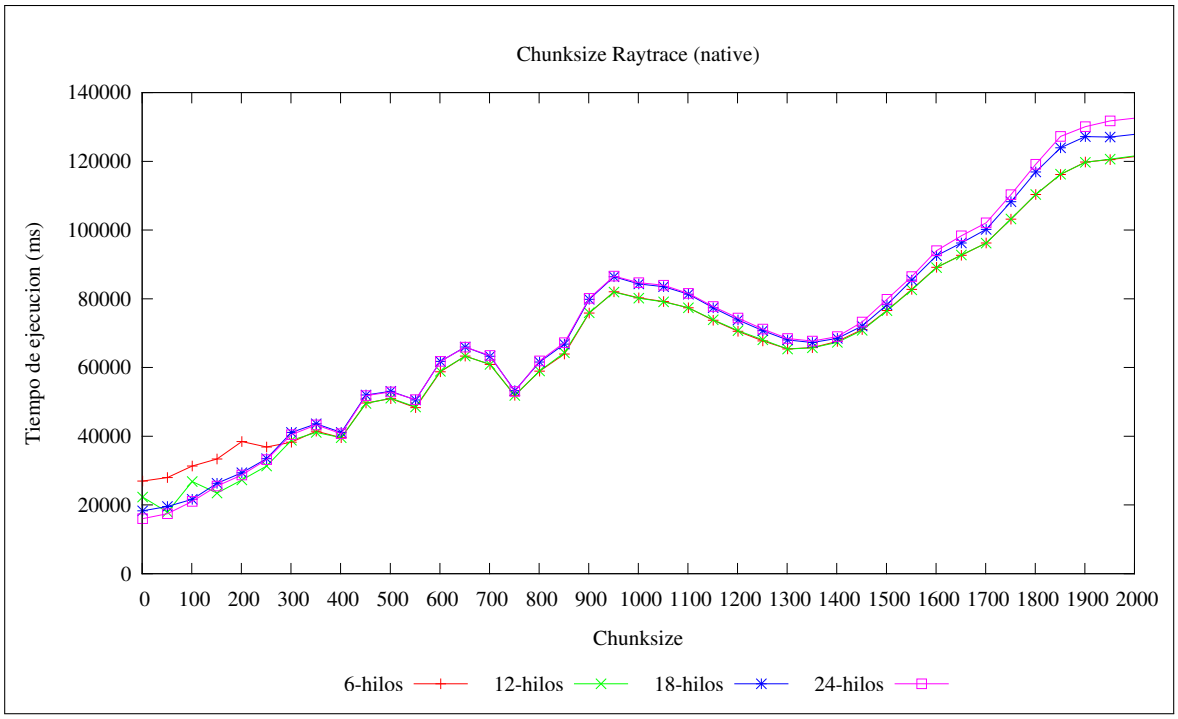


Figura 5-26: *Gráfica de tiempos de ejecución para los distintos valores del chunksize obtenidos a partir de la ejecución de la versión GrPPI Native de la aplicación Raytrace.*

Como se puede ver en la gráfica anterior los tiempos de ejecución van aumentando a medida que se aumenta el valor del *chunksize* salvo por algunas excepciones. Aunque podría valer tanto el valor 1 como el 50, en este caso se ha decidido usar el valor 50.

Una vez seleccionado el valor del *chunksize* se pasó a ejecutar la aplicación con los distintos modelos de programación que se iban a evaluar. Como ya se ha comentado se realizaron varias repeticiones de la ejecución obteniendo de esta forma varios tiempos de ejecución, a partir de los cuales se han obtenido diversas medidas de dispersión como son la media, varianza, desviación estándar e intervalos de confianza. En las siguientes tablas se pueden ver esas medidas de dispersión aplicadas a los tiempos resultantes de las ejecuciones con cada uno de los modelos de programación:

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
1	serial	121676	133,1746973	17735,5	$\mu \in$ [121578,0366 ; 121773,9634]
2	fastflow	63905,2	41,99642842	30,89261073	$\mu \in$ [62141,5 ; 65668,9]
2	pthreads	63781	47,14339827	2222,5	$\mu \in$ [63746,32127 ; 63815,67873]
2	*grppi_native*	64093,8	42,37570058	1795,7	$\mu \in$ [64062,6284 ; 64124,9716]
2	*grppi_openmp*	64118,6	118,9487285	14148,8	$\mu \in$ [64031,10121 ; 64206,09879]
2	*grppi_tbb*	68607,6	63,94763483	4089,3	$\mu \in$ [68560,56006 ; 68654,63994]

4	fastflow	34245,6	28,93613658	21,28544824	$\mu \in$ [33408,3 ; 35082,9]
4	pthreads	34096,4	22,16528818	491,3	$\mu \in$ [34080,09519 ; 34112,70481]
4	*grppi_native*	35278,8	43,39009103	1882,7	$\mu \in$ [35246,88221 ; 35310,71779]
4	*grppi_openmp*	34235,4	42,03926736	1767,3	$\mu \in$ [34204,47588 ; 34266,32412]
4	*grppi_tbb*	35747,2	26,78992348	717,7	$\mu \in$ [35727,49331 ; 35766,90669]
6	fastflow	23477,6	23,62837278	17,38105239	$\mu \in$ [22919,3 ; 24035,9]
6	pthreads	23353,8	16,30030675	265,7	$\mu \in$ [23341,80948 ; 23365,79052]
6	*grppi_native*	25513,6	53,93329213	2908,8	$\mu \in$ [25473,92662 ; 25553,27338]
6	*grppi_openmp*	23442,6	10,47854952	109,8	$\mu \in$ [23434,89197 ; 23450,30803]
6	*grppi_tbb*	23884	17,34935157	301	$\mu \in$ [23871,2378 ; 23896,7622]
8	fastflow	17695,8	16,60421633	12,21407655	$\mu \in$ [17420,1 ; 17971,5]
8	pthreads	17583,8	15,94365077	254,2	$\mu \in$ [17572,07184 ; 17595,52816]
8	*grppi_native*	21839,4	379,1527397	143756,8	$\mu \in$ [21560,4949 ; 22118,3051]
8	*grppi_openmp*	17656,4	12,3004065	151,3	$\mu \in$ [17647,35181 ; 17665,44819]
8	*grppi_tbb*	18325,6	27,22682501	741,3	$\mu \in$ [18305,57192 ; 18345,62808]
10	fastflow	14219	17,87456293	13,14854466	$\mu \in$ [13899,5 ; 14538,5]
10	pthreads	14118	4,847679857	23,5	$\mu \in$ [14114,43404 ; 14121,56596]
10	*grppi_native*	20216	210,4388272	44284,5	$\mu \in$ [20061,20101 ; 20370,79899]
10	*grppi_openmp*	14148,2	8,467585252	71,7	$\mu \in$ [14141,97124 ; 14154,42876]
10	*grppi_tbb*	14819,6	32,32336616	1044,8	$\mu \in$ [14795,8229 ; 14843,3771]
12	fastflow	11936,2	9,093954036	6,689520815	$\mu \in$ [11853,5 ; 12018,9]
12	pthreads	11810,2	6,90651866	47,7	$\mu \in$ [11805,11956 ; 11815,28044]
12	*grppi_native*	20522,6	251,5338943	63269,3	$\mu \in$ [20337,57144 ; 20707,62856]
12	*grppi_openmp*	11846,8	8,467585252	71,7	$\mu \in$ [11840,57124 ; 11853,02876]
12	*grppi_tbb*	12533,8	26,02306669	677,2	$\mu \in$ [12514,65741 ; 12552,94259]
14	fastflow	10665	20,68816087	15,21822985	$\mu \in$ [10237 ; 11093]
14	pthreads	10638	24,08318916	580	$\mu \in$ [10620,28438 ; 10655,71562]
14	*grppi_native*	24953,4	2615,781967	6842315,3	$\mu \in$ [23029,22842 ; 26877,57158]
14	*grppi_openmp*	11117	573,1539933	328505,5	$\mu \in$ [10695,3874 ; 11538,6126]
14	*grppi_tbb*	13212,4	554,7993331	307802,3	$\mu \in$ [12804,28911 ; 13620,51089]
16	fastflow	9499,2	17,97776404	13,22445949	$\mu \in$ [9176 ; 9822,4]
16	pthreads	9570,2	133,2186173	17747,2	$\mu \in$ [9472,204265 ; 9668,195735]
16	*grppi_native*	20245,2	560,4397381	314092,7	$\mu \in$ [19832,94002 ; 20657,45998]
16	*grppi_openmp*	9499,4	126,7884064	16075,3	$\mu \in$ [9406,134334 ; 9592,665666]
16	*grppi_tbb*	12065,6	200,721947	40289,3	$\mu \in$ [11917,94875 ; 12213,25125]
18	fastflow	8472,6	5,639148872	4,148163011	$\mu \in$ [8440,8 ; 8504,4]
18	pthreads	8495,8	49,39331939	2439,7	$\mu \in$ [8459,46623 ; 8532,13377]

18	*grppi_native*	19561,2	863,7868371	746127,7	$\mu \in$ [18925,79762 ; 20196,60238]
18	*grppi_openmp*	8555,4	113,6938873	12926,3	$\mu \in$ [8471,766674 ; 8639,033326]
18	*grppi_tbb*	11233	239,5360098	57377,5	$\mu \in$ [11056,79709 ; 11409,20291]
20	fastflow	7649,6	5,770615219	4,244869775	$\mu \in$ [7616,3 ; 7682,9]
20	pthreads	7710	98,39715443	9682	$\mu \in$ [7637,618964 ; 7782,381036]
20	*grppi_native*	18688,6	541,9523042	293712,3	$\mu \in$ [18289,93939 ; 19087,26061]
20	*grppi_openmp*	8251	782,7956311	612769	$\mu \in$ [7675,174826 ; 8826,825174]
20	*grppi_tbb*	11659,8	253,1110428	64065,2	$\mu \in$ [11473,61129 ; 11845,98871]
22	fastflow	6984,4	12,44186481	9,15224701	$\mu \in$ [6829,6 ; 7139,2]
22	pthreads	7198	123,9294154	15358,5	$\mu \in$ [7106,83741 ; 7289,16259]
22	*grppi_native*	18360	315,3109576	99421	$\mu \in$ [18128,05697 ; 18591,94303]
22	*grppi_openmp*	7518	520,1629552	270569,5	$\mu \in$ [7135,36766 ; 7900,63234]
22	*grppi_tbb*	11759,2	58,69156669	3444,7	$\mu \in$ [11716,02643 ; 11802,37357]
24	fastflow	6521,6	57,26080684	42,12110131	$\mu \in$ [3242,8 ; 9800,4]
24	pthreads	6657	40,0437261	1603,5	$\mu \in$ [6627,543799 ; 6686,456201]
24	*grppi_native*	17994,2	287,0168985	82378,7	$\mu \in$ [17783,07011 ; 18205,32989]
24	*grppi_openmp*	6410	24,82941804	616,5	$\mu \in$ [6391,735458 ; 6428,264542]
24	*grppi_tbb*	11750,4	86,61581842	7502,3	$\mu \in$ [11686,68533 ; 11814,11467]

Tabla 5.15: *Medidas de dispersión de los tiempos de ejecución obtenidos para la aplicación Raytrace.*

Esos tiempos medios que se pueden ver en las tablas, también se pueden ver representados en la siguiente gráfica, gracias a la cual se puede ver que la programación paralela mejora notablemente con respecto a la ejecución serial:

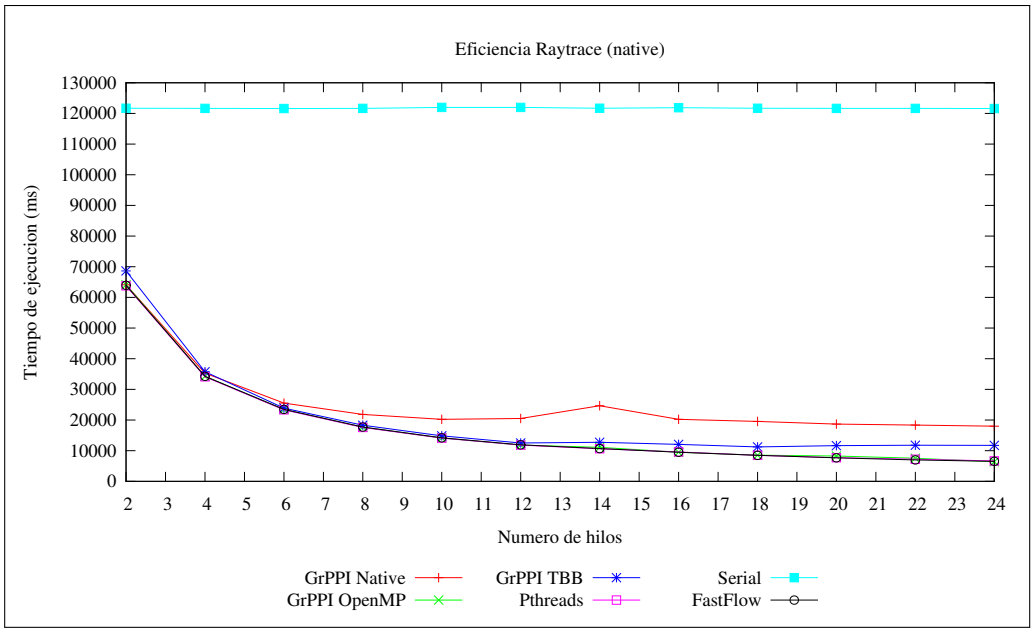


Figura 5-27: *Gráfica de tiempos de ejecución medios para los distintos modelos de programación de la aplicación Raytrace.*

Otra medida que se puede estudiar es el *speedup* o aumento de la eficiencia con respecto al modelo serial, lo cual se puede observar en la siguiente gráfica:

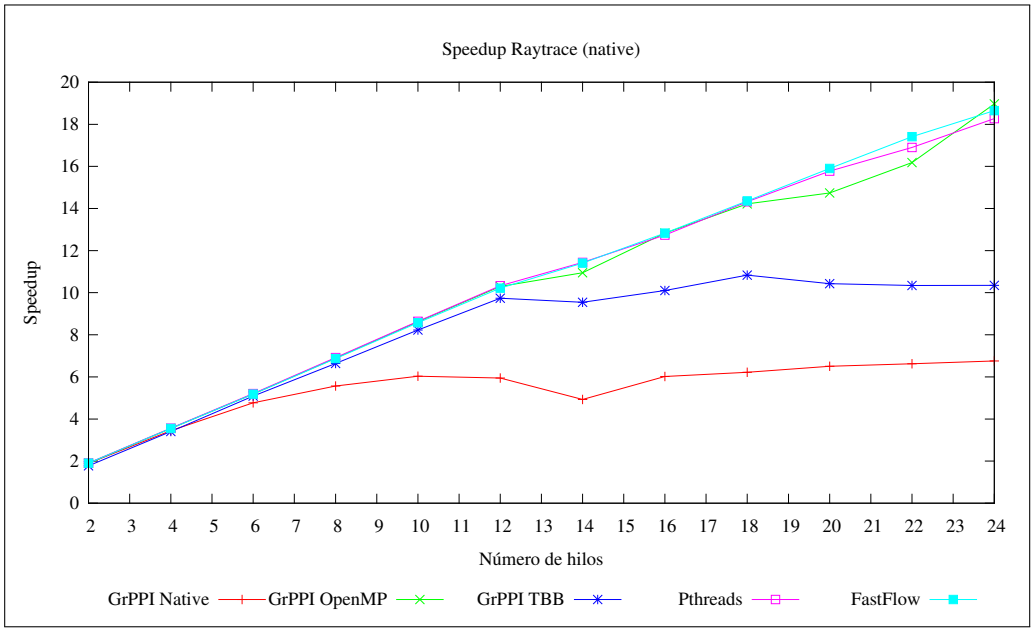


Figura 5-28: *Gráfica de los speedups para los distintos modelos de programación de la aplicación Raytrace.*

Como se puede ver en la primera gráfica, el tiempo de ejecución medio de los modelos paralelos es muy similar salvo en el caso de *GrPPI Native* que empieza a separarse a partir de la ejecución con 6 hilos y *GrPPI* cuyo tiempo medio comienza a aumentar a partir de la ejecución con 12 hilos.

Una vez descartados esos dos modelos, si se analiza la segunda gráfica, se puede ver como otros tres modelos tienen una escalabilidad bastante parecida excepto por ciertas bajadas para los modelos *GrPPI OpenMP y Pthreads* en la segunda mitad de la gráfica (de 12 a 24 hilos). De forma que a pesar de que cualquiera de los tres modelos podría ser eficaz en este caso por su constancia en cada una de las ejecuciones se podría decir que el mejor de los tres modelos sería *FastFlow*.

5.3.8. Streamcluster

En el caso del kernel *Streamcluster*, dependiendo del conjunto de entrada se modifican el número de puntos de entrada, el tamaño del bloque, el número de dimensiones de puntos dadas, el número de centros y el límite de centros intermedios que se pueden usar. Como en todas las aplicaciones y kernel ya estudiados el conjunto de entrada utilizado es el *native* que consta de 1.000.000 puntos de entrada, un tamaño de bloque de 200.000 puntos, las dimensiones de 128 puntos, de 10 a 20 centros y hasta 5.000 centros intermedios permitidos.

La implementación de este *kernel* implica el uso de patrones del tipo *parallel_for* lo que implica que se tuvo que realizar el estudio para determinar el mejor valor para el *chunksize*. Para este estudio se realizaron ejecuciones con 6, 12, 18 y 24 hilos, variando el valor del *chunksize* en 1 unidad en el rango entre 1 y el número de hilos. Los resultados obtenidos se muestran en la siguiente gráfica.

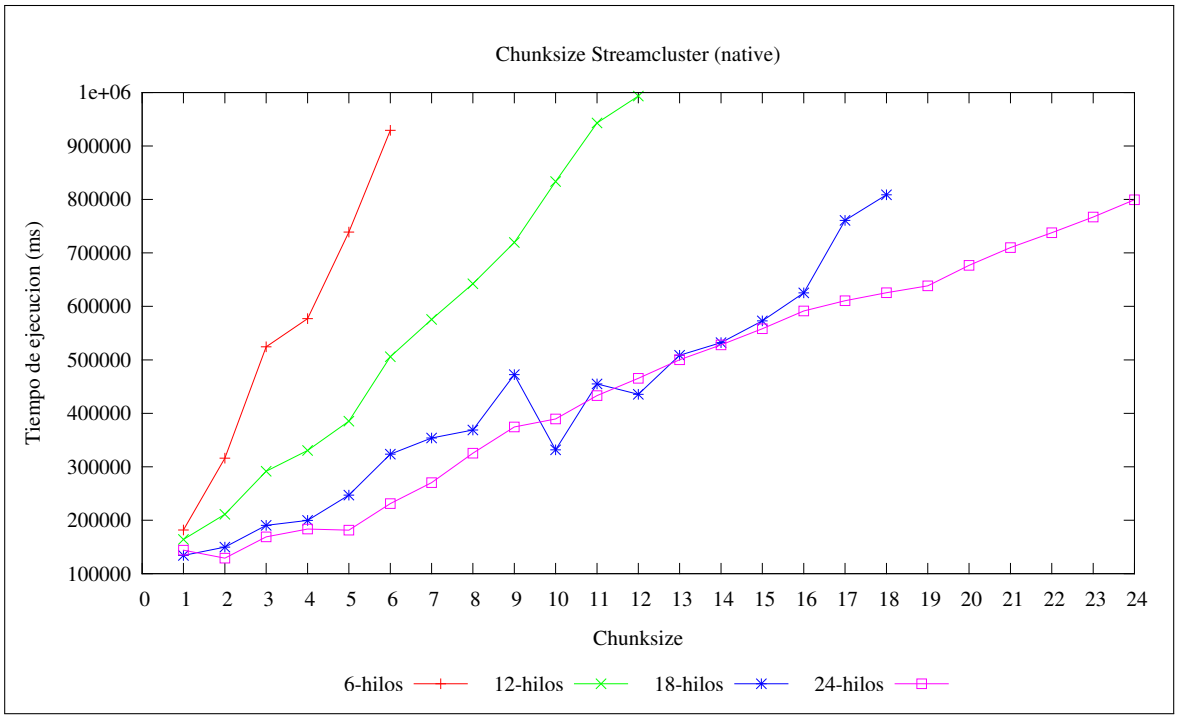


Figura 5-29: *Gráfica de tiempos de ejecución para los distintos valores del chunksize obtenidos mediante la implementación de GrPPI Native para el kernel Streamcluster.*

Como se puede observar en la gráfica anterior, todos los experimentos tienen su mínimo en el valor 1, por lo que dicho valor fue el escogido para el resto de las ejecuciones. En la siguiente tabla se muestran los resultados de las medidas de dispersión aplicadas sobre los tiempos de ejecución de cada uno de los modelos.

Hilos	Modelo	Media(μ) (ms)	Varianza(σ^2)	Desviación Típica(σ)	Intervalo de Confianza (90 %)
1	serial	927458	3520,223501	12391973,5	$\mu \in$ [924868,5204 ; 930047,4796]
2	fastflow	462025,4	99,52537365	9905,3	$\mu \in$ [461952,189 ; 462098,611]
2	pthreads	455327	867,8677895	753194,5	$\mu \in$ [454688,5957 ; 455965,4043]
2	*grppi_native*	425627,4	1901,739809	3616614,3	$\mu \in$ [424228,4785 ; 427026,3215]
2	*grppi_openmp*	424879	1465,23121	2146902,5	$\mu \in$ [423801,1746 ; 425956,8254]
2	*grppi_tbb*	422165,8	1822,954662	3323163,7	$\mu \in$ [420824,8329 ; 423506,7671]
4	fastflow	252733,2	1298,26931	1685503,2	$\mu \in$ [251778,1919 ; 253688,2081]
4	pthreads	233852,2	1177,382138	1386228,7	$\mu \in$ [232986,1166 ; 234718,2834]
4	*grppi_native*	240411,6	6651,399875	44241120,3	$\mu \in$ [235518,8242 ; 245304,3758]
4	*grppi_openmp*	235608,6	11212,06998	125710513,3	$\mu \in$ [27360,9912 ; 243856,2088]
4	*grppi_tbb*	243391,8	6641,788893	44113359,7	$\mu \in$ [238506,0941 ; 248277,5059]
6	fastflow	171981	614,1844186	377222,5	$\mu \in$ [171529,2054 ; 172432,7946]
6	pthreads	160733,4	423,2071597	179104,3	$\mu \in$ [160422,0884 ; 161044,7116]
6	*grppi_native*	181097,4	6427,741151	41315856,3	$\mu \in$ [176369,1478 ; 185825,6522]
6	*grppi_openmp*	155676	702,9601696	494153	$\mu \in$ [155158,9019 ; 156193,0981]
6	*grppi_tbb*	149628,8	12023,1779	144556806,7	$\mu \in$ [140784,5395 ; 158473,0605]
8	fastflow	132600,2	674,5407326	455005,2	$\mu \in$ [132104,0072 ; 133096,3928]
8	pthreads	122138,2	664,4597053	441506,7	$\mu \in$ [121649,4228 ; 122626,9772]
8	*grppi_native*	156715,8	5628,170191	31676299,7	$\mu \in$ [152575,7129 ; 160855,8871]
8	*grppi_openmp*	122441,2	2504,771287	6273879,2	$\mu \in$ [120598,688 ; 124283,712]
8	*grppi_tbb*	60276,2	67,85057111	4603,7	$\mu \in$ [60226,28906 ; 60326,11094]
10	fastflow	109435	287,1611046	82461,5	$\mu \in$ [109223,764 ; 109646,236]
10	pthreads	100018,4	324,2704118	105151,3	$\mu \in$ [99779,86639 ; 100256,9336]
10	*grppi_native*	178545,2	9114,177017	83068222,7	$\mu \in$ [171840,8031 ; 185249,5969]
10	*grppi_openmp*	96250,8	1799,432188	3237956,2	$\mu \in$ [94927,13605 ; 97574,46395]
10	*grppi_tbb*	98940	5342,265577	28539801,5	$\mu \in$ [95010,22461 ; 102869,7754]
12	fastflow	91338,2	291,735668	85109,7	$\mu \in$ [91123,59898 ; 91552,80102]
12	pthreads	85928	497,1277502	247136	$\mu \in$ [85562,31238 ; 86293,68762]
12	*grppi_native*	162828,6	9155,888996	83830303,3	$\mu \in$ [156093,5198 ; 169563,6802]
12	*grppi_openmp*	83645	3003,895221	9023386,5	$\mu \in$ [81435,33196 ; 85854,66804]
12	*grppi_tbb*	83131,2	2897,985801	8398321,7	$\mu \in$ [80999,43902 ; 85262,96098]

14	fastflow	80022,8	60,46651966	3656,2	$\mu \in$ [79978,32077 ; 80067,27923]
14	pthreads	79129,2	1157,056913	1338780,7	$\mu \in$ [78278,06789 ; 79980,33211]
14	*grppi_native*	167349,2	2994,365158	8966222,7	$\mu \in$ [165146,5423 ; 169551,8577]
14	*grppi_openmp*	63251,6	1556,487167	2422652,3	$\mu \in$ [62106,64663 ; 64396,55337]
14	*grppi_tbb*	68090	2333,971401	5447422,5	$\mu \in$ [66373,12853 ; 69806,87147]
16	fastflow	70622,8	145,4293643	21149,7	$\mu \in$ [70515,82203 ; 70729,77797]
16	pthreads	423200	1095,445115	1200000	$\mu \in$ [67910,38067 ; 69304,41933]
16	*grppi_native*	141526,8	4233,697875	17924197,7	$\mu \in$ [138412,488 ; 144641,112]
16	*grppi_openmp*	55947,2	2390,97432	5716758,2	$\mu \in$ [54188,39713 ; 57706,00287]
16	*grppi_tbb*	56261,6	3777,487101	14269408,8	$\mu \in$ [53482,87707 ; 59040,32293]
18	fastflow	64434,6	58,7137122	3447,3	$\mu \in$ [64391,41014 ; 64477,78986]
18	pthreads	60838,8	891,9580708	795589,2	$\mu \in$ [60182,67484 ; 61494,92516]
18	*grppi_native*	128899,2	8536,928587	72879149,7	$\mu \in$ [122619,4276 ; 135178,9724]
18	*grppi_openmp*	52812	3844,903055	14783279,5	$\mu \in$ [49983,68583 ; 55640,31417]
18	*grppi_tbb*	52217,8	4084,253383	16681125,7	$\mu \in$ [49213,41952 ; 55222,18048]
20	fastflow	59541,2	285,0906873	81276,7	$\mu \in$ [59331,48703 ; 59750,91297]
20	pthreads	55985,2	344,2233287	118489,7	$\mu \in$ [55731,98901 ; 56238,41099]
20	*grppi_native*	123540,4	3217,058641	10349466,3	$\mu \in$ [121173,9288 ; 125906,8712]
20	*grppi_openmp*	46041,6	3077,266287	9469567,8	$\mu \in$ [205177,2513 ; 205342,3487]
20	*grppi_tbb*	48811,8	3555,962978	12644872,7	$\mu \in$ [46196,03042 ; 51427,56958]
22	fastflow	55735,8	277,1483357	76811,2	$\mu \in$ [55531,92943 ; 55939,67057]
22	pthreads	53398,4	474,5969869	225242,3	$\mu \in$ [53049,28603 ; 53747,51397]
22	*grppi_native*	141060,4	12708,56681	161507670,3	$\mu \in$ [131711,9668 ; 150408,8332]
22	*grppi_openmp*	43468	4310,088572	18576863,5	$\mu \in$ [40297,49495 ; 46638,50505]
22	*grppi_tbb*	44277,8	3380,303936	11426454,7	$\mu \in$ [41791,24537 ; 46764,35463]
24	fastflow	49633,6	72,00902721	5185,3	$\mu \in$ [49580,63009 ; 49686,56991]
24	pthreads	50378,6	1525,390048	2326814,8	$\mu \in$ [49256,5217 ; 51500,6783]
24	*grppi_native*	142435,2	11935,99299	142467928,7	$\mu \in$ [133655,0728 ; 151215,3272]
24	*grppi_openmp*	42238,6	2737,418821	7493461,8	$\mu \in$ [40224,95224 ; 44252,24776]
24	*grppi_tbb*	40976,2	1862,69501	3469632,7	$\mu \in$ [39605,99987 ; 42346,40013]

Tabla 5.16: *Medidas de dispersión de los tiempos de ejecución obtenidos para el kernel Streamcluster.*

A continuación, se muestran los tiempos de ejecución medios para cada uno de los modelos de programación usados en forma de gráfica para una mejor evaluación de los resultados.

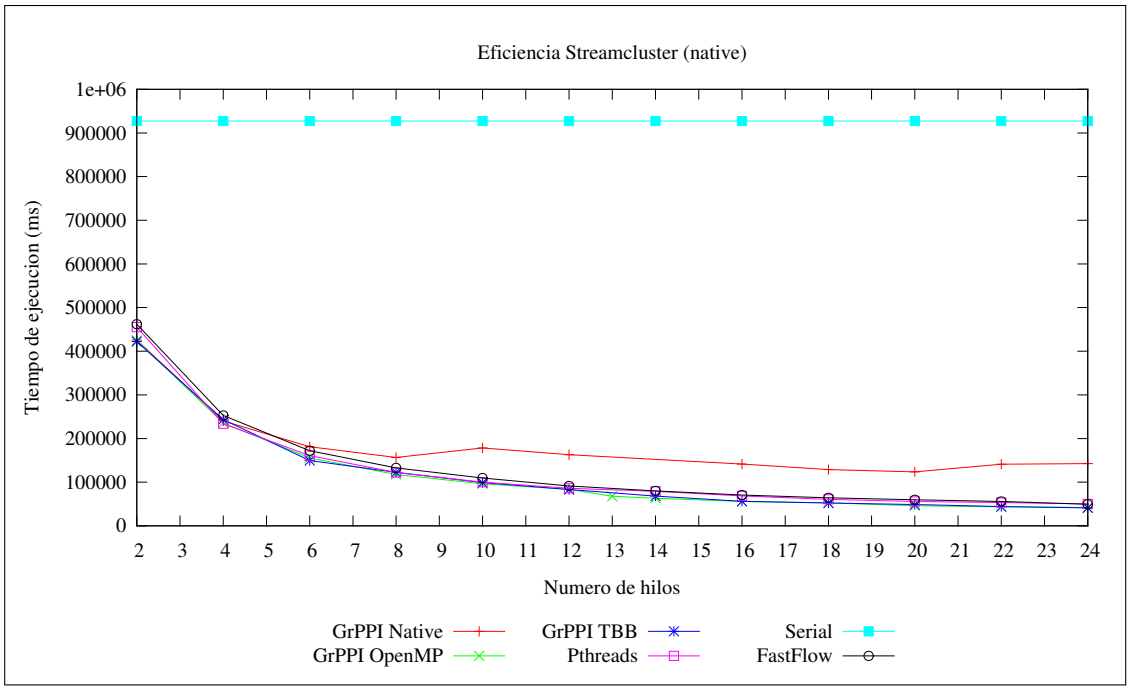


Figura 5-30: *Gráfica de tiempos de ejecución medios para los distintos modelos de programación del kernel Streamcluster.*

A parte del tiempo de ejecución medio, también se podría considerar otra medida para la evaluación de los modelos, esta medida es el *speedup* o aumento de la eficiencia con respecto a la versión serial, lo cual se puede observar en la siguiente gráfica:

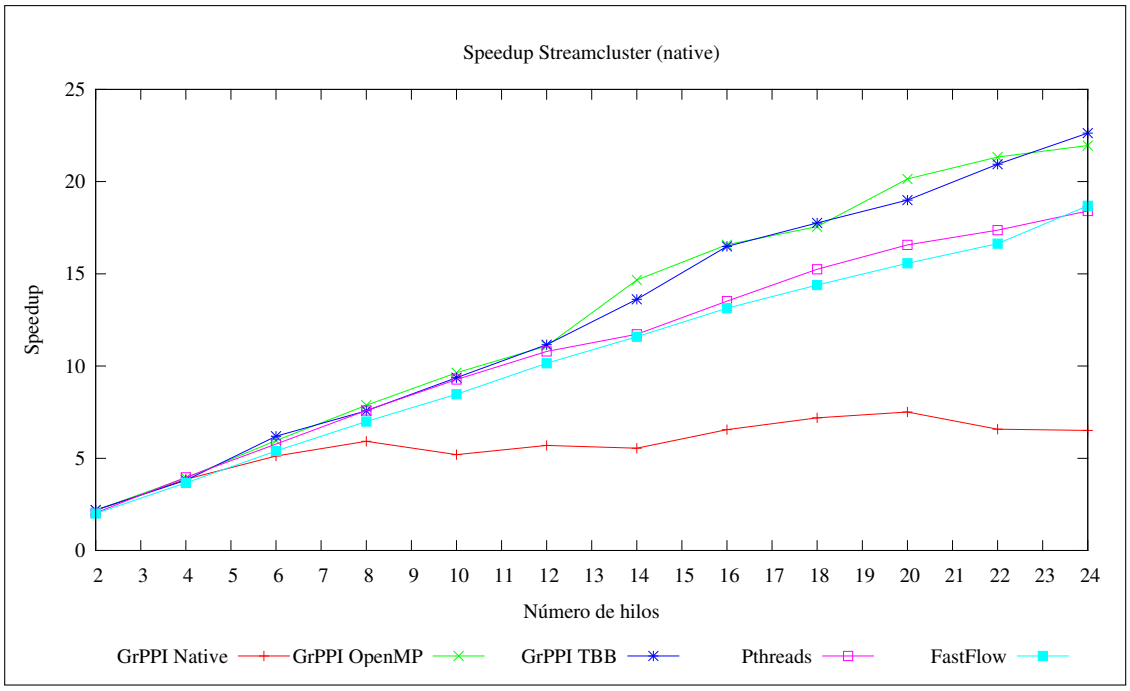


Figura 5-31: *Gráfica de los speedups para los distintos modelos de programación del kernel Streamcluster.*

Como se puede ver en la primera gráfica, la línea correspondiente con el modelo *GrPPI Native* a partir de 6 hilos se estabiliza y deja de escalar mientras que el tiempo de ejecución para el resto de los modelos sigue disminuyendo con el aumento del número de hilos, pero no se distingue cual disminuye en mayor medida. Analizando la segunda gráfica, se ve claramente que los dos mejores modelos son *GrPPI OpenMP y GrPPI TBB*.

Aunque si se quiere establecer un único modelo como el mejor se puede pasar a analizar el *speedup* medio de ambos modelos.

Modelo	Speedup Medio
GrPPI_TBB	12,563
GrPPI_OpenMP	12,742

Tabla 5.17: *Speedups medios para los modelos con mejor rendimiento para el kernel Streamcluster.*

Como se puede observar en la tabla anterior, el modelo con mayor valor medio del *speedup* es *GrPPI OpenMP* de forma que dicho modelo es el más adecuado para paralelizar esta aplicación.

Una vez se han analizado cada una de las aplicaciones y *kernels* seleccionados, se puede hacer un resumen de los mejores modelos para cada uno de los casos. En siguiente tabla se muestra el modelo que se considera como el mejor entre las implementaciones de las que ya constaba el *benchmark*, es decir, sin tener en cuenta los resultados obtenidos para las implementaciones de *GrPPI*, el mejor modelo cuando se analiza también las versiones de *GrPPI* y, por último, el porcentaje de mejora entre los modelos establecidos como mejores en ambos casos.

Aplicación	Mejor Modelo sin GrPPI	Mejor Modelo con GrPPI	Porcentaje de mejora
Blackscholes	*OpenMP*	*OpenMP*	0 %
Bodytrack	*TBB*	*TBB*	0 %
Canneal	*FastFlow*	*GrPPI OpenMP*	14,48 %
Facesim	*FastFlow*	*GrPPI OpenMP*	18,32 %
Ferret	*TBB*	*TBB*	0 %
Fluidanimate	*FastFlow*	*FastFlow*	0 %
Raytrace	*FastFlow*	*FastFlow*	0 %
Streamcluster	*Pthreads*	*GrPPI OpenMP*	15,62 %

Tabla 5.18: *Tabla resumen con los mejores modelos de programación paralela para cada una de las aplicaciones o kernels analizados.*

Capítulo 6

Marco Regulador

En este capítulo se hace mención a la normativa técnica y las restricciones de uso del software empleado. En concreto se expone un análisis legislativo (Sección 6.1) y un análisis de los estándares técnicos aplicables (Sección 6.2).

En el análisis legislativo se tiene en cuenta tanto las leyes aplicables como las restricciones del software que se ha utilizado para el desarrollo del trabajo de fin de grado. Mientras que en la sección de estándares técnicos se describen brevemente los estándares del lenguaje de programación utilizado (C++) y el estándar de calidad del software.

6.1. Análisis Legislativo

Este trabajo de fin de grado está enfocado en la evaluación del rendimiento de distintos modelos de programación sobre un conjunto de aplicaciones, por lo que lo primero a tener en cuenta es que tanto los modelos de programación como todas las aplicaciones usadas sean de carácter público.

6.1.1. Licencias

En cuanto a los modelos de programación, se han utilizado tres modelos básicos de programación de código abierto como son *OpenMP, TBB y Pthreads* y otros dos modelos no tan comúnmente utilizados como son *FastFlow* y *GrPPI*, ambos de código abierto también.

- **Pthreads**: Como se puede ver en la página web de GNU [34], este modelo es de código abierto bajo la licencia GFDL 1.2 [35] y Copyright ©2010, 2012, 2013 Free Software Foundation, Inc.

- **OpenMP**: Como se puede ver en el artículo publicado para la versión 4.5 [36], este modelo tiene Copyright ©1997-2015 OpenMP Architecture Review Board, el cual permite la copia y uso de OpenMP siempre que se ponga el aviso de copyright y se nombre su artículo.

- **TBB**: Como se puede ver en su página web [37], este modelo consta de una licencia dual, compuesta por una licencia comercial (COM) y otra *open source* bajo la licencia Apache v2.0 de enero de 2004, la cual se puede consultar en su página web [38].

- **FastFlow**: Como se puede ver en su página web [39], este es un modelo de código abierto que se puede usar bajo la licencia LGPLv3 [40] o GNU GPL [41].

- **GrPPI**: Este es un modelo creado por el grupo ARCOS de la Universidad Carlos III de Madrid, como se puede ver en su repositorio [31] es de código abierto bajo la licencia de Apache [42] y Copyright ©2018 Universidad Carlos III de Madrid.

Por otro lado, en cuanto a las aplicaciones utilizadas para la evaluación, todas provienen del *benchmark* PARSEC que es de código abierto como se puede ver en su página web [43], siempre que se cite su artículo resumen sobre el *benchmark* [32].

Este *benchmark* está sujeto al Copyright ©2006-2009 Princeton University y su licencia permite su redistribución y uso tanto con cambios como sin ellos siempre que se cumplan las siguientes condiciones:

- Las redistribuciones del código fuente deben conservar el aviso de copyright anterior, esta lista de condiciones y la siguiente exención de responsabilidad.

- Las redistribuciones en formato binario deben reproducir el aviso de copyright anterior, esta lista de condiciones y la siguiente exención de responsabilidad en la documentación y / u otros materiales proporcionados con la distribución.

- Ni el nombre de la Universidad de Princeton ni los nombres de sus colaboradores se pueden utilizar para respaldar o promocionar productos derivados de este software sin un permiso previo específico por escrito.

THIS SOFTWARE IS PROVIDED BY PRINCETON UNIVERSITY “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL PRINCETON UNIVERSITY BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF

LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

A parte de la licencia general del *benchmark*, hay ciertas aplicaciones y bibliotecas que están sujetas a otras licencias (solo se van a nombrar las de las aplicaciones que se han usado en este proyecto).

- *Bodytrack*: tanto el código fuente como las entradas de esta aplicación están sujetos a la licencia Apache v2.0 [38], al igual que el modelo TBB.
- *Facesim y Canneal*: el código fuente y las entradas de estas aplicaciones están sujetos a la propia licencia de PARSEC (*BSD-style license*).
- *Ferret*: el código fuente de esta aplicación está sujeta a la licencia GNU GPL versión 3 [44], mientras que las entradas son de dominio público.
- *GSL*: esta biblioteca se usa en la aplicación *Ferret* y está sujeta a la licencia GNU GPL versión 2 [45].
- *Mesa*: esta biblioteca se usa en la aplicación *Raytrace*, está bajo el Copyright ©1999-2007 Brian Paul y varias licencias como se puede ver en su página web [46], aunque la principal es la licencia MIT.

El software que se ha desarrollado durante este trabajo de fin de grado se va a distribuir bajo las mismas condiciones que se establecen en la licencia de P^3ARSEC y cada una de las licencias asociadas a las aplicaciones usadas.

Una vez analizadas las licencias de los modelos de programación y aplicaciones, se deben estudiar los datos que usan las aplicaciones seleccionadas y sobre todo hay que asegurarse de que no sean de carácter personal. Para ello se ha tenido en cuenta la ley orgánica 15/1999 de protección de datos de carácter personal establecida por el gobierno de España [47]. Sin embargo, en este caso ninguna de las aplicaciones seleccionadas está usando datos potencialmente sensibles o de carácter personal, por lo que la ley de protección de datos mencionada no se puede aplicar.

6.2. Estándares Técnicos

En esta sección se describen todos los estándares relevantes, en este caso dichos estándares son los relativos al lenguaje de programación utilizado para la implementación del proyecto.

6.2.1. Estándar de C++ ISO/IEC 14882

Los estándares ISO (*International Organization for Standarization*) e IEC (*International Electrotechnical Comission*) establecen la solución a un problema particular a la cual se ha llegado mediante un consenso global. Cada uno de ellos es un comité independiente y se unen solo cuando se trata de un campo de interés común.

Se pueden encontrar varias ediciones del estándar de C++ como se puede ver en la Figura 6-1, pero en este caso se van a comentar la tercera edición de 2011 y la cuarta de 2014.

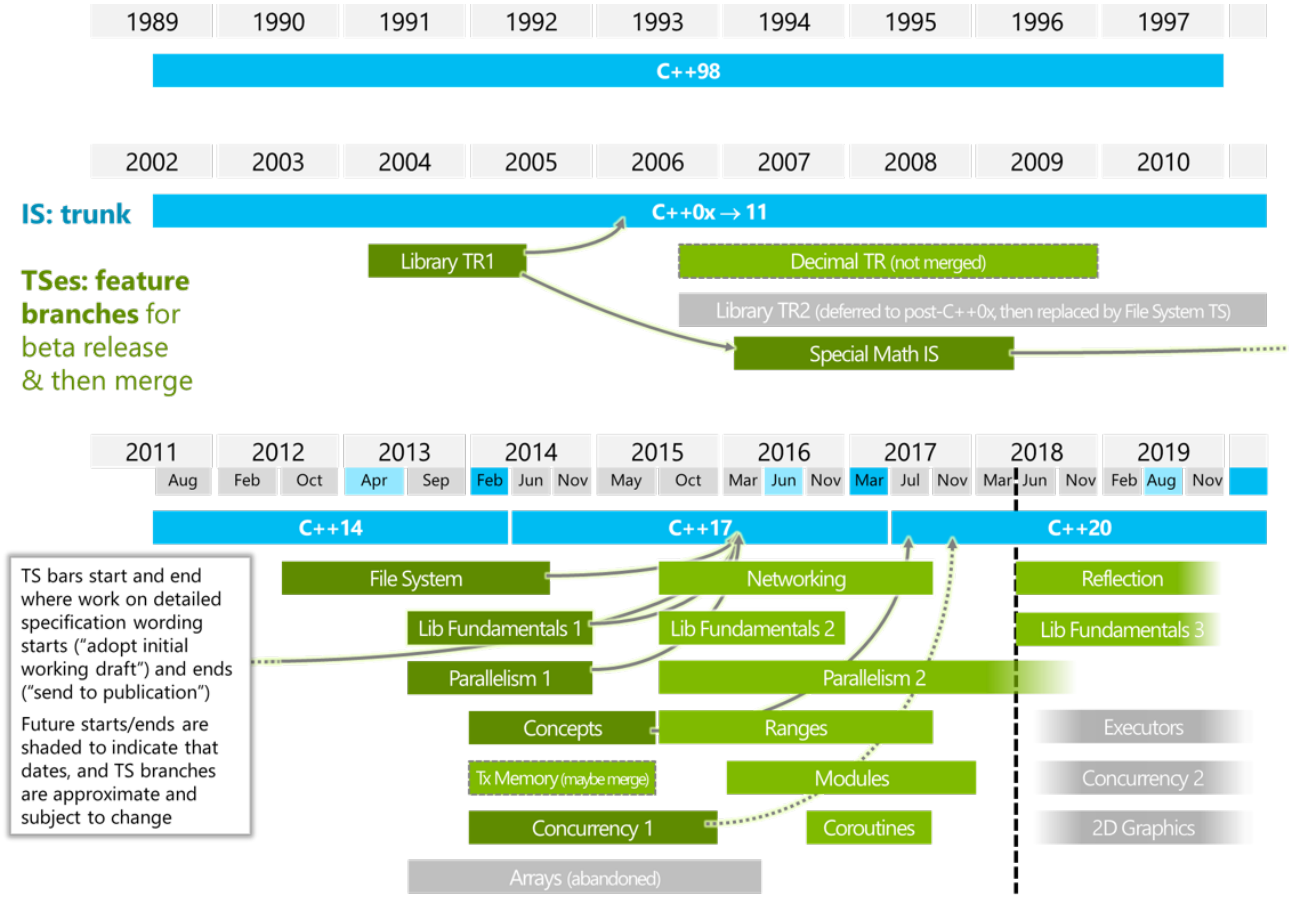


Figura 6-1: *Evolución del estándar de C++ en cada una de las ediciones.* [fuente: `https://isocpp.org/std/status`]

2011

C++11 es la versión del estándar para el lenguaje de programación C++ aprobada por los comités ISO e IEC el 12 de agosto de 2011 [5], remplazando a la anterior versión de 2003. En esta versión se introdujeron mejoras en varios campos: el soporte multihilo (*multithreading*), la programación genérica, la inicialización

uniforme y el rendimiento.

2014

C++14 fue la versión del estándar que remplazó a la de 2011, esta se aprobó el 18 de agosto de 2014 [48]. Esta versión incorporó corrección de errores, se extendieron algunas funciones de las incorporadas en la versión de 2011 y se añadieron otras nuevas.

Capítulo 7

Planificación

Este proyecto fue planificado por tareas, es decir, se dividió el trabajo en tantas tareas como fuesen necesarias para llevarlo a cabo, con un tiempo estimado para cada una de ellas.

Se van a mostrar dos diagramas de *Gantt*:

1. El que se elaboró al inicio para organizar el tiempo del proyecto.
2. El real con todos los tiempos empleados en cada una de las tareas.

A continuación, se muestra la descripción de las tareas en las que se ha dividido el proyecto:

- **TAREA 1:** *Instalación y configuración del entorno*, esta tarea consiste en descargar de forma local el código fuente de varios de los modelos de programación que se iban a emplear durante el desarrollo del proyecto. En concreto *GrPPI, FastFlow y TBB*, ya que tanto ISO C++ Threads como OpenMP viene implícitos con la versión del compilador, lo que explica la última subtarea.
  - **Tarea 1.1:** GrPPI
  - **Tarea 1.2:** FastFlow
  - **Tarea 1.3:** TBB
  - **Tarea 1.4:** Actualizar la versión del compilador (4.3.4 o superior)
- **TAREA 2:** *Aprendizaje de los modelos de programación basados en patrones paralelos*, al ser la primera vez que se trataba con varios de los modelos de programación, se tuvo que dedicar un poco de tiempo al aprendizaje de estos.
  - **Tarea 2.1:** GrPPI

  - **Tarea 2.2:** FastFlow

- **TAREA 3:** P^3ARSEC *Benchmark*, desde un principio se sabía que intentar abordar todas las aplicaciones del *benchmark* era demasiado trabajo por ello se decidió dedicar un periodo de tiempo al análisis de los programas disponibles y así elegir los más interesantes para la evaluación.
  - **Tarea 3.1:** Instalación y configuración del Benchmark
  - **Tarea 3.2:** Análisis y selección de las aplicaciones y kernels a evaluar
- **TAREA 4:** Aplicación Blackscholes
  - **Tarea 4.1:** Análisis de la aplicación
  - **Tarea 4.2:** Implementación
  - **Tarea 4.3:** Evaluación de los resultados obtenidos
- **TAREA 5:** Aplicación Bodytrack
  - **Tarea 5.1:** Análisis de la aplicación
  - **Tarea 5.2:** Implementación
  - **Tarea 5.3:** Evaluación de los resultados obtenidos
- **TAREA 6:** Aplicación Canneal
  - **Tarea 6.1:** Análisis de la aplicación
  - **Tarea 6.2:** Implementación
  - **Tarea 6.3:** Evaluación de los resultados obtenidos
- **TAREA 7:** Aplicación Facesim
  - **Tarea 7.1:** Análisis de la aplicación
  - **Tarea 7.2:** Implementación
  - **Tarea 7.3:** Evaluación de los resultados obtenidos
- **TAREA 8:** Aplicación Ferret
  - **Tarea 8.1:** Análisis de la aplicación
  - **Tarea 8.2:** Implementación
  - **Tarea 8.3:** Evaluación de los resultados obtenidos
- **TAREA 9:** Aplicación Fluidanimate
  - **Tarea 9.1:** Análisis de la aplicación

  - **Tarea 9.2:** Implementación
  - **Tarea 9.3:** Evaluación de los resultados obtenidos

- **TAREA 10:** Aplicación Raytrace
  - **Tarea 10.1:** Análisis de la aplicación
  - **Tarea 10.2:** Implementación
  - **Tarea 10.3:** Evaluación de los resultados obtenidos

- **TAREA 11:** Aplicación Streamcluster
  - **Tarea 11.1:** Análisis de la aplicación
  - **Tarea 11.2:** Implementación
  - **Tarea 11.3:** Evaluación de los resultados obtenidos

- **TAREA 12:** *Obtención de los tiempos de ejecución*, una vez se había terminado de implementar todas las aplicaciones seleccionadas y que funcionarán de manera correcta, se podía pasar a la obtención de tiempo de ejecución. Para ello hacía falta en primer lugar una forma de medir esos tiempos y la manera más sencilla y rápida era mediante la colocación de *timers* antes y después del cómputo del programa. Después de colocarlos ya simplemente se tenían que realizar las ejecuciones, pero como eran demasiadas para hacerlas a mano se crearon *scripts* de ejecución. Y, por último, una vez que se habían obtenido los tiempos se podían esbozar las gráficas de eficiencia y *speedup*.
  - **Tarea 12.1:** Colocación de los *timers*
  - **Tarea 12.2:** Implementación y ejecución de los *scripts*
  - **Tarea 12.3:** Esbozado de gráficas

- **TAREA 13:** Redacción de la memoria

- **TAREA 14:** Revisión de la memoria y corrección de errores

Una vez se han expuesto las tareas y subtareas necesarias se puede pasar a la presentación del primer *Gantt*, en el cual solo se representarán las tareas (de la 1 a la 14), ya que si se representaran también las subtareas sería demasiado grande. Este primer *Gantt* como ya se ha comentado es el que se desarrolló inicialmente con los tiempos estimados de las tareas:

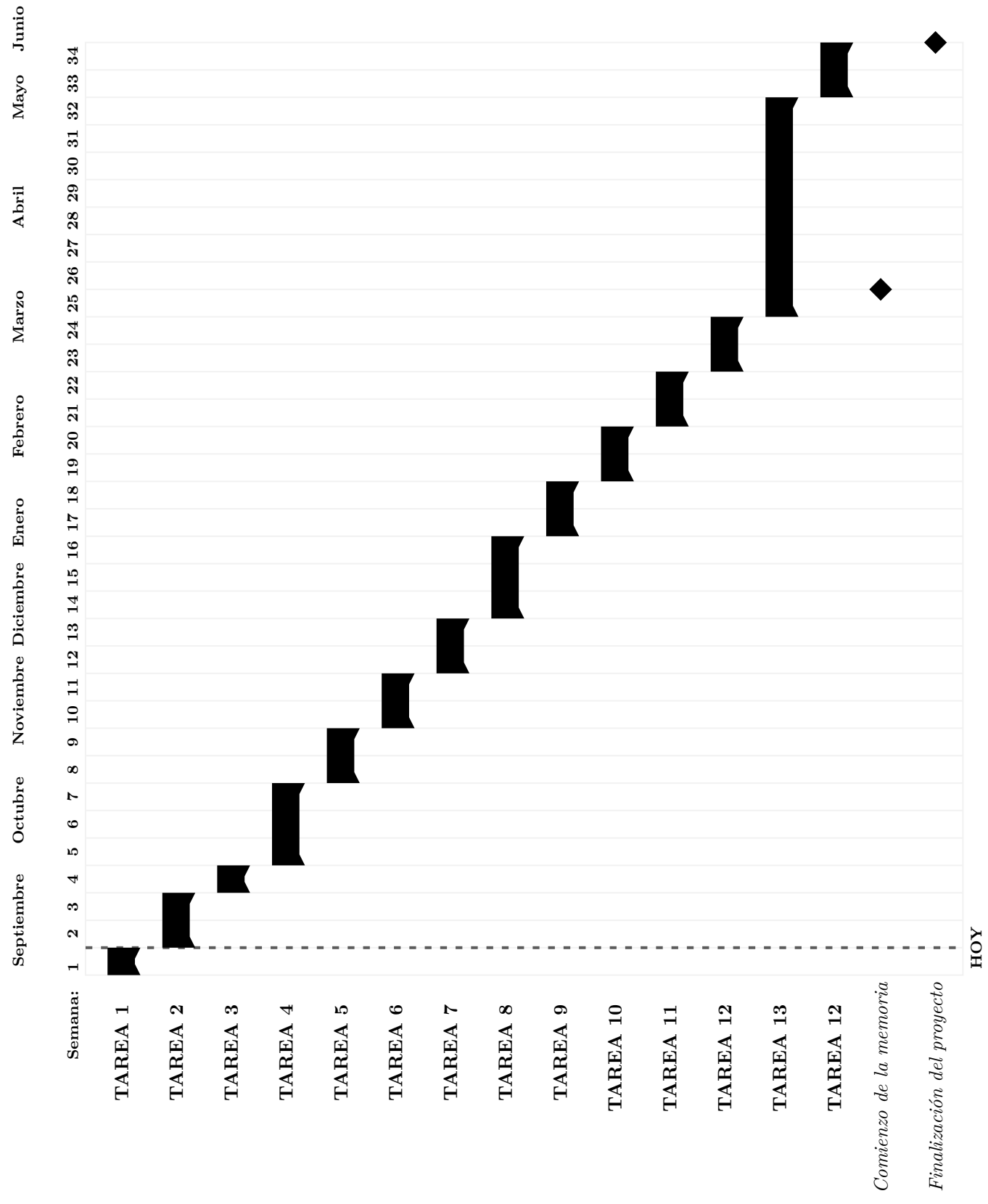


Figura 7-1: *Diagrama de Gantt estimado al inicio del proyecto*

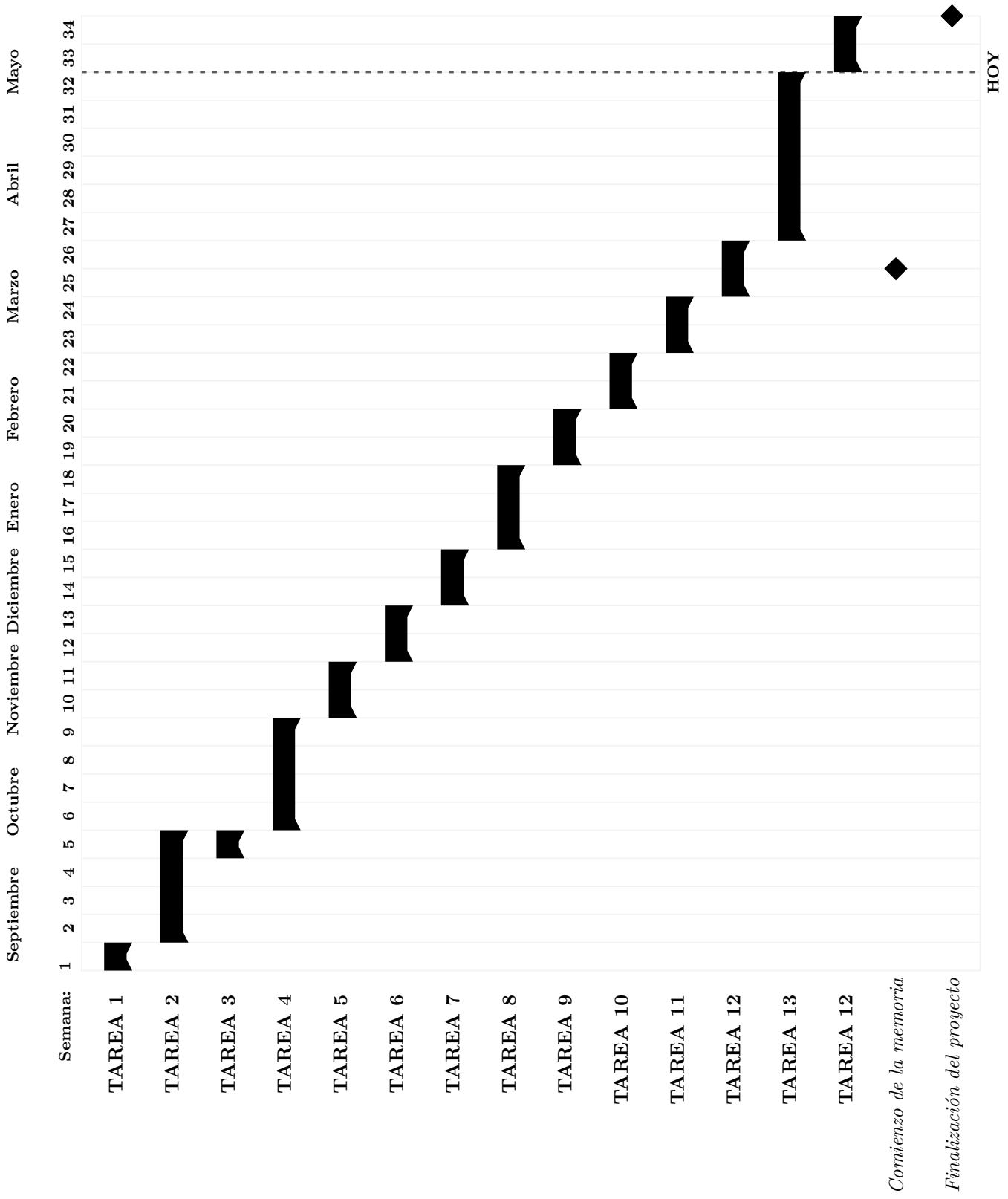


Figura 7-2: *Diagrama de Gantt real del proyecto una vez terminado*

A pesar de tener una planificación estricta para la realización de cada tarea, algunas de ellas llevaron más tiempo de lo que se había planificado en un principio, por lo que el Diagrama 7-1 se tuvo que modificar, dando lugar al Diagrama 7-2.

Como se puede ver en el Diagrama 7-2 la tarea 2 llevó más tiempo de lo planificado, ya que al ser la primera vez que se trataba con ese tipo de modelo de programación no fue tan sencillo comprender el funcionamiento.

Otra tarea que llevó más tiempo del estimado en primera instancia fue la tarea 4, es decir, la de implementación de la primera aplicación del *benchmark*, esto se debió a que aún no se conocía el funcionamiento del *benchmark*: el modo de ejecución, los archivos que se creaban al hacerlo, la organización, etc.

Estos cambios supusieron un retraso en el resto de las tareas que llevó a tener que realizar dos tareas al mismo tiempo, en concreto la tarea 2 y la tarea 3, y aun así el tiempo que en un principio se había destinado para realizar la memoria se vio disminuido ya que se tuvo que empezar a escribir una semana después de lo planificado, como se puede ver en el Diagrama 7-2.

Capítulo 8

Entorno Socio-Económico

En este capítulo se tratan los aspectos socioeconómicos que engloban el proyecto, por un lado, en la Sección 8.1 se desglosan los precios de los elementos involucrados en el desarrollo del proyecto y, por otro lado, en la Sección 8.2 se explica el impacto que tiene este trabajo de fin de grado en el ámbito económico.

8.1. Presupuesto

En esta sección se va a realizar una descripción detallada del coste estimado necesario para llevar a cabo este proyecto. Como regla general para elaborar un presupuesto de un proyecto se deben tener en cuenta distintos costes como los de los recursos humanos, los materiales empleados: donde se incluye el hardware, software y los consumibles, los costes indirectos, etc.

8.1.1. Costes de Recursos Humanos

Para determinar el coste de los recursos humanos, es decir el personal que ha desarrollado el proyecto se contabilizan las horas que han sido necesarias para la realización del mismo. Como se puede ver en el Capítulo 7 han sido necesarias 34 semanas y una media de 20 horas semanales, lo que hace un total de 680 horas. Para definir el coste por hora se ha tomado como base la cotización a la Seguridad Social de un ingeniero técnico establecida por el Ministerio de Empleo y Seguridad Social, Régimen General [49].

Número de Horas	€/Hora	Total (€)
680	13,5	9.180

Tabla 8.1: *Costes de Recursos Humanos*

8.1.2. Costes de Recursos Materiales

Los costes materiales engloban todo lo relacionado con el hardware y software utilizados para el desarrollo del trabajo de fin de grado y los consumibles.

Hardware

En cuanto al coste del hardware hay que tener en cuenta tanto el servidor que se ha usado para realizar las pruebas, como el ordenador personal en el que se ha desarrollado la implementación. Como se ha comentado en la Sección 5.1 el servidor usado ha sido una plataforma con arquitectura NUMA, cuyo coste viene definido por el importe de cada uno de sus componentes.

Elemento	Coste (€)	Vida útil (meses)	Coste Mensual (€)	Meses de uso	Coste Amortizado (€)	
Intel Xeon Ivy Bridge	2.011 x 2	48	83,8	10	838	[50]
RAM 128GB DDR3	1.129	48	23,52	10	235,2	[51]
Ordenador Personal	720	48	15	10	150	[52]
Total					1.223,2	

Tabla 8.2: *Coste del Hardware*

Software

En cuanto al software se debe considerar el sistema operativo tanto del servidor como del ordenador personal (Ubuntu 14.04.2 LTS), todos los modelos de programación que se han empleado (OpenMP, TBB, FastFlow, GrPPI, PThreads, ISO C++ Threads), el *benchmark* utilizado para la evaluación (P^3ARSEC) y el sistema de composición de textos utilizado para realizar la memoria (LaTeX).

En este caso, todos los recursos software ya mencionados son de código abierto y por lo tanto no suponen ningún coste adicional para el presupuesto.

Consumibles

Los consumibles se refieren a todos los costes relacionados con el papel, cartuchos de tinta, bolígrafos, etc.

Concepto	**Coste (€)**
Consumibles	100

Tabla 8.3: *Coste de los Consumibles*

8.1.3. Costes Indirectos

Los costes indirectos son los relacionados con el proceso de producción: la electricidad, el teléfono, el internet, climatización, etc. En general, se suele establecer como coste indirecto el 20 % del coste de los recursos humanos.

Concepto	**Coste (€)**
Costes Indirectos (20 %)	1.836

Tabla 8.4: *Costes Indirectos*

8.1.4. Coste Total

La siguiente tabla muestra el coste total del proyecto obtenido mediante la suma de los costes anteriores y las tasas.

Concepto	**Coste (€)**
Recursos Humanos	9.180
Hardware	1.223,2
Software	0
Consumibles	100
Costes Indirectos	1.836
Beneficios (10 %)	1.233,92
Total sin impuestos	13.573,12
Impuestos (21 %)	2.850,35
Total	16.423,47

Tabla 8.5: *Coste Total*

8.2. Impacto Socio-Económico

Este proyecto al ser de investigación no tiene como resultado un software de carácter comercial, simplemente pretende evaluar distintas opciones ya existentes para la aplicación de computación paralela en situaciones reales.

En cuanto al impacto social, este proyecto pretende incentivar un incremento en el uso de la programación paralela, al verificarse en la evaluación realizada que los resultados son realmente buenos, llegando en algunos casos a disminuir el tiempo de ejecución en un 92 % con respecto a la ejecución secuencial pues se

explotan en mayor medida todos los recursos hardware de los que consta la máquina.

Aun siendo un software de carácter no comercial, éste también conlleva un impacto económico ya que al usar computación paralela se disminuye la cantidad de recursos energéticos necesarios, provocando de esta forma una disminución del coste económico total para realizar una misma tarea.

Capítulo 9

Conclusiones

En este trabajo de fin de grado, se han llevado a cabo diversos análisis empleando modelos de programación paralela de bajo nivel (*ISO C++ Threads, Pthreads*), de alto nivel (*OpenMP, TBB*) y basados en patrones paralelos (*FastFlow, GrPPI*) con el objetivo de determinar cuál es el modelo más eficiente dentro de los estudiados. Se seleccionaron ocho aplicaciones dentro del *benchmark* P^3ARSEC con una gran variedad de dominios, tipos de datos, operaciones a realizar, etc, de forma que se pudiera estudiar si había algún modelo óptimo para todos los casos por muy diferentes que éstos fueran.

Una vez realizada la evaluación (Sección 5.3), se ha podido concluir que no hay un modelo que sea claramente mejor que todos los demás, sino que el modelo a elegir depende de las características de la aplicación sobre la que se va a aplicar, como se puede ver en la tabla resumen (Tabla 5.18) que se presenta al final de dicha sección. Por lo tanto, para poder paralelizar una aplicación de la manera más eficiente posible es necesario realizar un análisis previo con los modelos de programación paralela potenciales y determinar el más adecuado para ese caso concreto.

En las siguientes secciones se tratan los objetivos que se han cumplido con la realización de este proyecto, así como las líneas futuras de trabajo que se podrían seguir para continuar con la línea de este proyecto.

9.1. Objetivos Cumplidos

En esta sección se analiza que objetivos de los establecidos en la Sección 1.2 se han cumplido durante el desarrollo del proyecto.

- El primer objetivo consistía en escoger un *benchmark* de evaluación de aplicaciones paralelas ampliamente aceptado que permitiera comparar los resultados de los distintos modelos de programación

paralela y escoger entre sus aplicaciones las más relevantes, teniendo en cuenta que debían abarcar distintos dominios de actuación con el fin de ser lo más representativo posible.

En cuanto a la selección del *benchmark*, en la Sección 2.3 se muestra el análisis que se ha llevado a cabo con el objetivo de elegir el *benchmark* que mejor se adaptara a las especificaciones del proyecto y finalmente se ha optado por el *benchmark* P^3ARSEC.

En cuanto a la elección de las aplicaciones, en la Sección 4.2 se describen cada una de las aplicaciones que se eligieron finalmente para realizar la evaluación, lo que corrobora que efectivamente pertenecen a dominios diferentes.

- El segundo objetivo consistía en desarrollar versiones para cada una de las aplicaciones seleccionadas que expresaran el paralelismo mediante el uso de patrones de diseño paralelos y concretamente mediante el uso del *framework GrPPI*.

  En la Sección 4.2 se muestra la implementación realizada para cada una de las partes paralelizables de las aplicaciones usando dicha biblioteca de paralelismo.

- El tercero consistía en realizar una evaluación del rendimiento que permitiera comparar el impacto que tenían los modelos de programación paralela básicos frente a aquellos basados en patrones de diseño paralelos.

  En la Sección 5.3 se realiza dicho análisis, determinando a partir de los resultados obtenidos en cuanto a tiempo de ejecución y *speedup* el modelo más adecuado para usar en cada uno de los casos.

9.2. Líneas Futuras de Trabajo

Las líneas futuras de trabajo que se podrían seguir son:

- Evaluar las nuevas implementaciones de *GrPPI*. Como se comentó en la Sección 2.2.6, este modelo de programación paralela se basa en diversas implementaciones internas de otros modelos diferentes con el objetivo de evaluarlos de forma sencilla. En este trabajo de fin de grado no se han evaluado todas las implementaciones de las que dispone actualmente *GrPPI* por falta de tiempo, como se indicó en la Sección 4.1, por ello una de estas líneas futuras de trabajo es precisamente evaluar la implementación de *FastFlow* que no se ha podido abordar en este proyecto.

- Extender el proyecto para evaluar no solo modelos basados en CPU, sino también modelos basados en GPU (CUDA, OpenCL, etc.).

Capítulo 10

English Summary

10.1. Introduction

In recent years, there has been an increase of computation needs, that is, an increment in the complexity and workload of the programs, which has led to the development of new techniques capable of decreasing the execution time needed and raise their efficiency and their throughput. The HPC (High-Performance Computing) is the study of those techniques. It includes multicore architectures being able of running beyond 1TFLOP and parallel computing.

The multicore architectures came up due to that necessity of increasing the computation power and the fact that the techniques that had been using until that moment weren't feasible as it represented an enormous growth of energy needs. Therefore, from that point on researchers started increasing the number of cores composing the system instead of the transistors' number, giving room to the multicore architectures.

These multicore architectures can also have several sockets leading to multi-socket architectures, such as NUMA (Non-Uniform Memory Access), which is the one used for the evaluation phase. Each of the compounding sockets can access any of the cores and their associated memory space, but accessing the non-local memory it isn't as cheaper, in terms of time, as accessing the local memory.

The multicore architectures that belongs to HPC are the supercomputers and clusters. These are a good method of increasing the efficiency when running a program but consumes a huge amount of energy, such is the case that some of the current energy plants are not able to produce it. This fact has led to the researchers to look for cheaper alternatives, such as parallel computing.

Parallel computing is a programming technique which allows the simultaneous execution of several code sections in order to decrease the total execution time. This type of computing is based on the principle

that every problem can be divided into several subproblems which can be run concurrently as long as they are not dependent on each other. Nevertheless, using parallel computing isn't as carefree as sequential programming due to the errors that can entail such as deadlocks, data races, overhead, starvation, etc, which are usually because of mistakes in the communication and synchronization of the threads.

The parallel computing can be done by means of threads managed manually or using parallel patterns which internally deal with such problems avoiding that the user has to worry about them.

There are three types of parallelism:

- The *data parallelism* consists of doing the same task over different non-dependent datasets.
- The *task parallelism* consists of different cores applying diverse tasks over a dataset.
- The *streaming parallelism* is slightly dissimilar to the previous ones as it occurs when the input arrives continuously, that is, that the dataset has not a fixed size and it is not entirely available since the beginning.

Michael J.Flynn established the first computer architecture classification based on the parallelism distribution aforementioned. That standard divides the computer architectures into the following four types:

- **SISD** *(Single Instruction Single Data)*: this type of architecture is purely sequential as it just executes one instruction over a dataset on each clock tick.
- **SIMD** *(Single Instruction Multiple Data)*: this kind of architecture exploits the data parallelism as each of the cores compounding the system executes the same instruction over a dataset concurrently.
- **MISD** *(Multiple Instruction Single Data)*: this class of architecture is used only for theoretical purposes as it produces redundant parallelism.
- **MIMD** *(Multiple Instruction Multiple Data)*: this type of architecture consists of multiple cores executing concurrently different instructions over distinct datasets, exploiting task parallelism.

There are several parallel programming models, APIs and parallel programming languages developed specifically for implementing programs able to run on top of multicore architectures. This bachelor's thesis focuses on the parallel programming models, distinguishing three types: low-level models, high-level models and parallel-pattern-based models.

10.1.1. Objective

Given the growing need to parallelize the programs in order to obtain better results in terms of performance, the use of parallel design patterns shows up as a solution to express the parallelism of the applications in

a simple way that allows a better maintenance of the same.

In this context, the aim of this paper is to verify the performance impact of using parallel design patterns to express applications. To achieve this objective, the performance of a well-known set of applications of the benchmark will be evaluated by comparing the performance of the use of patterns with other alternatives.

This objective is materialized in the following specific objectives:

- Choose a benchmark evaluation of parallel applications widely accepted and that allows comparing the results of different models of parallel programming. Within this benchmark, the most relevant applications for the evaluation will be selected, taking into account that they must be from different domains of action to achieve a more representative analysis.
- Develop versions of selected applications that express parallelism in terms of parallel design patterns using the GrPPI framework.
- Perform a performance evaluation of selected applications that allows comparing the impact of different programming models against the use of parallel design patterns.

10.2. Results of the Evaluation

As said before, this bachelor's thesis is focused on parallel programming models, which can be defined as a set of abstractions giving the programmer a simplified vision of the architecture and facilitating the reference between logical and physical components, exploiting in a greater extent the parallelism.

There is a great variety of parallel programming models but this thesis is just focused on C++ programming-based ones and specifically on *Pthreads, ISO C++ Threads, TBB, OpenMP, FastFlow and GrPPI.*

ISO C++ Threads

C++ Threads come defined by *thread* class, which is available since C++11. Using this type of threads, you can carry out concurrently several processes. They start the execution of the associated function code just after the creation of the object.

Their termination can follow two patterns depending on the associated synchronization option: if the option join is used means that each thread has to wait until the rest end; on the contrary, if the option detach is used means that the thread is independent of the rest so it will terminate just when it finished the execution.

Pthreads

Pthreads or POSIX Threads is a set of interfaces similar to the previous model, but instead of being defined by the class *thread*, these ones are defined by the class *pthreads*. This library is a standards-based API for C/C++ programming languages.

Pthreads library creates and terminates threads by means of the following functions: *pthreads_create, pthread_join and pthread_exit.*

Regarding the synchronization, just like the previous model, it needs the use of mutexes and semaphores in order to manage the synchronization and communication between threads.

TBB

TBB is an Intel designed library used along with C++ programming language for parallel computing with shared-memory and heterogeneous computing.

This model has several data structures and functions that allow the user to avoid bugs derived from system dependent activities such as creation, communication, synchronization and destruction of threads. That fact is one of the main features of TBB library, as it allows the separation between the programming task and the characteristics of the device.

TBB implements tasks stealing in order to balance the thread workload so that the system exploitation and scalability increases.

OpenMP

OpenMP (Open Multi-Processing) is an API which allows the multicore programming with shared-memory over multiple platforms. It is based on several directives and pragmas that can be applied over iterative code blocks.

Due to its shared-memory characteristic, the variables used during the execution of the program are shared between all the threads involved even if this fact can produce data races. In order to avoid that data races, OpenMP implements primitives capable of specifying whenever a variable is shared or private.

This API has three different scheduling policies: if the policy is static means that each thread will execute the same amount of chunks; if it is dynamic means that the chunks are scheduled by demand, that is, whenever a thread finished its execution a new chunk is assigned to it; and if it is guided means that the scheduling task start using large blocks and its size decrease over time, so each time a thread finished its execution takes a new chunk until all of them are processed.

ISO C++ Threads and Pthreads belongs to the low-level model type, while *TBB and OpenMP* belong to high-level models.

Regarding the other kind of models, it should be pointed out that they are not based on primitives or functions as the previous ones, instead of that, they are based on parallel patterns, which are high-level structures that automatically communicate and synchronize threads, making easier the programmer task.

There are several types of parallel patterns which are classified depending on the kind of parallelism that they exploit.

Data Parallel Patterns

- The *Map* pattern applies the same instruction to each of the elements of an input list simultaneously, generating finally a new output list.
- The *Parallel_for* pattern is quite similar to the previous one, but instead of iterating between the elements of an input list, it iterates over the values within a range and applies over each the same instruction.
- The *Reduce* pattern applies the same instruction over each pair of elements of an input list, storing those partial results and combining them later to obtain the final result.
- The *Fork* pattern applies different operations over each of the elements of an input list.
- The *Map_Reduce* pattern arises from the combination of Map and Reduce patterns. First, it applies the map pattern over each of the elements of an input list and then the reduce pattern over the resulting elements of the previous pattern.
- The *Stencil* pattern applies the same instruction to each of the elements of an input list and its neighbourhood.

Task Parallel Patterns

- The *Divide & Conquer* pattern allows the continuous division of a complex problem into simpler subproblems until those resulting subproblems can be solved easily.
- The *Branch & Bound* pattern divides recursively the search space and extracts the resultant elements out of the subspaces by means of an objective function.

Streaming Parallel Patterns

- The *Pipeline* pattern is composed of at least two stages: the generation phase in which the elements are created and an intermediate phase in which the operations are applied and if there is any subsequent phase it passes the result. Optionally, it can also have a consumption phase which is similar to the intermediate ones but it not return any value.

- The *Farm* pattern allows the application of instructions over several data elements using threads. This pattern can be used jointly to other patterns such as pipeline.
- The *Filter* pattern is similar to *if* statement as it filters the elements maintaining only those ones that satisfy the predicate.
- The *Iteration* pattern is similar to while statement as it produces a loop which applies operations on each data element until the predicate is satisfied.
- The *Reduction* pattern is similar to the data reduce parallel pattern but instead of being applied to a fixed length list, it is applied to a data stream.

Once explained the type of parallel patterns that exist, GrPPI and Fastflow parallel pattern based models can be described.

FastFlow

FastFlow is a parallel programming framework based on C++ templates and implemented over Pthreads library. It was created for its use over parallel heterogeneous platforms and specifically shared-memory clusters.

It has three levels: the lower one implements a lock-free queue with a consumer and a producer; the second level implements queues with one producer and several consumers and queues with several producers and one consumer; and the last level which provides the parallel patterns.

Regarding data parallel patterns, FastFlow consist of the following ones: ParallelFor (parallel_for), ParallelForReduce (parallel_reduce) and ParallelForPipeReduce (parallel_reduce_idx). And regarding streaming parallel patterns, it consists of the next ones: Pipeline (ff_Pipe) and Farm (ff_Farm).

GrPPI

GrPPI is a library implemented using C++ templates which allows in a simple way the evaluation of several parallel programming models. This model as opposed to the previous one, not only provide parallel patterns but also different internal implementations based on other parallel programming models such as OpenMP, TBB, ISO C++ Threads and FastFlow. In spite of having that four versions, this bachelor's thesis is focused only on the three first ones.

When defining a parallel pattern you need to specify the model you want to use for the execution, so changing between one model an another is quite easy. This characteristic makes GrPPI a good model for doing an evaluation.

GrPPI consists of the following parallel patterns: Map, Reduce, Map_Reduce, Stencil, Divide &

Conquer, Pipeline, Farm, Stream Filter, Steam Iteration and Stream Reduction. The definition of these patterns is just the same that the one explained before.

In order to evaluate each of the aforementioned parallel programming models, is necessary to choose a benchmark. A benchmark is a set of applications selected on purpose for efficiency evaluation. There are several benchmarks composed of parallel applications that can be valid in this case, such as SPEC, NPB, STAP, SPLASH, Rodinia and PARSEC, but the one selected is P3ARSEC.

P3ARSEC is a variation of the PARSEC benchmark that includes apart from the original PARSEC implementation of the applications, the FastFlow version.

It consists of nine applications such as Blackscholes, Bodytrack, Facesim, Ferret, Fluidanimate, Freqmine, Raytrace, Swaptions, Vips and x264, and three kernels such as Canneal, Dedup and Streamcluster. Each of those programs also has six different input sets (test, simdev, simsmall, simmedium, simlarge and native) in order to add diversity, but this bachelor's thesis has focused on the native sets since are the recommended ones for throughput and efficiency evaluation. Due to time lack, only eight of these programs have been analyzed during the evaluation, specifically, these are Blackscholes, Bodytrack, Canneal, Facesim, Ferret, Fluidanimate, Raytrace and Streamcluster.

Blackscholes

Blackscholes is an Intel RMS benchmark application employed for financial analysis in order to compute the prices for a European options portfolio by means of applying the Black-Scholes equation. This application has implementations for the following models:

- In the case of *Pthreads*, as many threads as stated in the input are created. The portfolio size is divided by the number of threads created and the resulting value is the number of options that are assigned to each thread.
- In the case of *OpenMP*, its primitive *#pragma omp parallel for* is used. Additionally, some of the variables needed for the computation are stated as private in order to avoid the data races.
- In the case of *TBB*, its functions *tbb::split, tbb::blocked_range, tbb::affinity_partitioner* and *tbb::parallel_for* are used.
- In the case of *FastFlow*, its parallel pattern *parallel_for* is used, but not the general type, the *parallel_for_thid* one as it uses the thread identification number in order to assign to each of them the proper division of the portfolio.
- In the case of *GrPPI*, two implementations have been made: the first one make use of the parallel_for pattern and the second one uses two nested patterns, a farm with a map inside.

After the Blackscholes evaluation, it is concluded the model that gives the maximum scalability for this application is OpenMP. Additionally, the experiment proved that the *GrPPI parallel_for* version obtains better results in terms of time than the *farm_map* one.

Bodytrack

Bodytrack is an application used in computer vision and pattern recognition fields in order to track the 3D pose of a human body using a set of images taken with several cameras simultaneously.
This application has implementations for the following models:

- In the case of *Pthreads*, it implements an additional code file called WorkPoolThread.h, as it uses a pool of threads for doing the computation in parallel.
- In the case of *OpenMP*, its primitive *#pragma omp parallel for* is used in order to parallelize each of the *for* loops.
- In the case of *TBB*, its functions *tbb::blocked_range* and *tbb::parallel_for* are used along with the override of the () operator.
- In the case of *FastFlow*, all the potentially parallelizable loops are rewritten in order to fit with the *parallel_for* pattern.
- In the case of *GrPPI*, it has tried to make an implementation equivalent to the *FastFlow* one, so the *parallel_for* pattern has also used for the *GrPPI* implementation.

After the Bodytrack evaluation, it is concluded the model that gives the maximum scalability for this application is TBB.

Canneal

Canneal is a kernel developed by Princeton University for engineering field as it tries to find the lower routing cost for chips design using the simulated annealing method with cache recognition.
This application has implementations for the following models:

- In the case of *Pthreads*, it creates as many threads as stated in the input, so that each of them is in charge of doing the exchanges and routing cost computation.
- In the case of *FastFlow*, it is used a *Farm* parallel pattern in which each of the threads that carry out the exchanges and calculation of the routing cost is workers in the farm pattern.
- In the case of *GrPPI*, instead of using the Farm pattern as in the *FastFlow* implementation, it uses the *parallel_for* pattern in order to create each of the threads.

After the Canneal evaluation, it is concluded the model that gives the maximum scalability for this kernel is GrPPI OpenMP.

Facesim

Facesim is an Intel RMS benchmark application developed by Stanford University for the animation field. It takes a human face as model and tries to simulate its movements as realistic as possible by means of the underlying physics.
This application has implementations for the following models:

- In the case of *Pthreads*, several additional files are implemented in order to allow the use of the thread pool for the parallel execution of three processes involved in the computation.
- In the case of *FastFlow*, several *parallel_for* patterns are implemented in order to exploit the task parallelism.
- In the case of *GrPPI*, the pattern *parallel_for* is used for the implementation so that it is equivalent to the *FastFlow* version.

After the Facesim evaluation, it is concluded the model that gives the maximum scalability for this application is GrPPI OpenMP.

Ferret

Ferret is an application developed by Princeton University for the content-based similarity search field, in order to do that it uses feature-rich data such as audio, video, images, etc. This application can be divided into six phases: Input, Image segmentation, Feature extraction, Indexing, Classification and Output. Its aim consists of returning images similar to the query one.
This application has implementations for the following models:

- In the case of *Pthreads*, it creates as many threads as stated in the input and assigns them to the phase they are going to carry out. It also creates a queue for each phase including the input and output ones.
- In the case of *TBB*, it uses its functions *tbb::filter(SEQUENTIAL_OUT_OF_ORDER_FILTER)* for the sequential phases, input and output and *tbb::filter(parallel)* for the intermediate ones.
- In the case of *FastFlow*, there are four implementations: the first one using the pattern *Farm*, the second one using also the pattern *Farm* but optimizing the intermediate code and the other two using nested patterns, the third one uses a *Pipeline* pattern with *Farms* inside and the last one uses a *Farm* with a *Pipeline* inside.
- In the case of *GrPPI*, it implements the same four versions as FastFlow.

After the Ferret evaluation, it is concluded the model that gives the maximum scalability for this application is TBB. Additionally, the experiment proved that the GrPPI farm_optimized version obtains better results in terms of time than the rest of GrPPI versions.

Fluidanimate

Fluidanimate is an Intel RMS benchmark application developed for the animation field. It uses an extension of the smoothed particle hydrodynamic method along with the Navier-Stokes equations in order to simulate the movements of a Newtonian fluid.
This application has implementations for the following models:

- In the case of *Pthreads*, it creates as many threads as stated in the input and assigns each of them to the complete cycle of phases.
- In the case of *TBB*, it uses its function *tbb::task* as the type of each function in the application, and the functions *tbb::task::spawn_root_and_wait* and *tbb::task::allocate_root* for the parallization.
- In the case of *FastFlow*, it implements several times its *parallel_for* pattern.
- In the case of *GrPPI*, it implements nine patterns of *parallel_for* type.

After the Fluidanimate evaluation, it is concluded the model that gives the maximum scalability for this application is FastFlow.

Raytrace

Raytrace is an Intel RMS benchmark application developed for the rendered field. It uses the raytracing method in order to generate realistic 3D images.
This application has implementations for the following models:

- In the case of *Pthreads*, it creates as many threads as stated in the input using the function *createThreads*, each of them is initialized by means of the *startThreads* function and synchronized by means of the *waitForAllThreads* method.
- In the case of *FastFlow*, it uses the *parallel_for* pattern in order to divide the resolution stated in the input and assign to each thread the proper fraction.
- In the case of *GrPPI*, it implements an equivalent version to the FastFlow one, so it uses the *parallel_for* pattern.

After the Raytrace evaluation, it is concluded the model that gives the maximum scalability for this application is FastFlow.

Streamcluster

Streamcluster is an RMS kernel developed by Princeton University for the data mining field. It tries to solve the online clustering problem.
This application has implementations for the following models:

- In the case of *Pthreads*, it creates as many threads as stated in the input using the function *pthread_create* and then all are synchronized using barriers.
- In the case of *TBB*, it uses its functions *tbb::task::spawn_root_and_wait, tbb::task::allocate_root, tbb::blocked_range, tbb::parallel_reduce, tbb::parallel_for, tbb::task_list* and *tbb::task*.
- In the case of *FastFlow*, it uses several times the *parallel_for* pattern and once the *parallel_for_reduce* one.
- In the case of *GrPPI*, it implements an equivalent version to the FastFlow one, so it uses seven times the *parallel_for* pattern and once the *reduce* pattern.

After the Streamcluster evaluation, it is concluded the model that gives the maximum scalability for this kernel is GrPPI OpenMP.

10.3. Conclusions

In this bachelor's thesis, it has been done several experiments, which has consisted on carrying out executions of different applications using low-level parallel programming models (Pthreads, ISO C++ Threads), high-level (OpenMP, TBB) and parallel patterns-based (FastFlow, GrPPI) in order to determine which of those is the best model. In order to do that, eight applications were chosen within the P^3ARSEC benchmark taking into account that they had to be of a great variety of domains, data types, operations to be performed, etc, so that after the experiments could answer the question asking if there is a model which is the best for all the cases.

Once the evaluation has been carried out (Section 5.3), it has been possible to conclude that there is no model that is clearly better than all the others and that the model has to be chosen according to the application characteristics, as it can be seen in the summary table (Table 5.18) presented at the end of that section. Therefore, in order to parallelize an application in the most efficient way possible, it is necessary to carry out a preliminary analysis with the potential parallel programming models and determine the more suitable for that particular case.

10.3.1. Achieved Objectives

As discussed in Section 10.1.1, the main objective of this bachelor's thesis has been divided into five tasks, which have been carried out throughout the development of the project as explained below.

- The first objective was to choose a widely accepted parallel application evaluation benchmark that would allow comparing the results of the different parallel programming models and choose among

their most relevant applications, taking into account that they had to cover different domains of action in order to be as representative as possible.

Regarding the selection of the benchmark, Section 2.3 shows the analysis that has been carried out with the objective of choosing the benchmark that best suits the specifications of the project and finally has opted for the P3ARSEC benchmark.

Regarding the choice of applications, Section 4.2 describes each of the applications that were finally chosen to perform the evaluation, which corroborates that they effectively belong to different domains.

- The second objective was to develop versions for each of the selected applications that expressed parallelism through the use of parallel design patterns and specifically through the use of the GrPPI framework.

  Section 4.2 shows the implementation performed for each of the parallelizable parts of the applications using said parallelism library.

- The third was to perform a performance evaluation that would allow comparing the impact of basic parallel programming models against those based on parallel design patterns.

  In Section 5.3, this analysis is carried out, determining from the results obtained in terms of execution time and speedup the most appropriate model to use in each of the cases.

Acrónimos

AMD *Advanced Micro Devices*

API *Application Programming Interface*

ARCOS *ARquitectura de COmputadoreS*

ASSIST *A Software development System based upon Integrated Skeleton Technology*

AVC *Advanced Video Coding*

BASH *Bourne-Again SHell*

BHV *Bounding Volume Hierarchy*

CPU *Central Processing Unit*

CUDA *Compute Unified Device Architecture*

CNDF *Cumulative Normal Distribution Function*

FIMI *Frequent Itemset Mining Implementations*

FP-growth *Frequent Pattern growth*

GPU *Graphics Processing Unit*

GRPPI *Generic Reusable Parallel Pattern Interface*

GSL *GNU Scientific Library*

HPC *High Performance Computing*

IBM *International Business Machines*

IEC *International Electrotechnical Commission*

IEEE *Institute of Electrical and Electronics Engineers*

ISO *International Organization for Standardization*

LSH *Locality-Sensitive Hashing*

MIMD *Multiple Instruction Multiple Data*

MISD *Multiple Instruction Single Data*

MPI *Message Passing Interface*

MUESLI *MUEnster Skeleton LIbrary*

NPB *NAS Parallel Benchmark*

NUMA *Non-Uniform Memory Access*

OPENACC *Open ACCelerators*

OPENCL *Open Computing Language*

OPENMP *Open Multi-Processing*

PARSEC *Princeton Application Repository for Shared-Memory Computers*

P^3ARSEC *Parallel Pattern PARSEC*

PTHREADS *POSIX Threads*

PFLOPS *Peta Floating-Point Operations Per Second*

RAM *Random Access Memory*

RMS *Recognition, Mining and Synthesis*

RT-STAP *Real-Time Space-Time Adaptive Processing*

SA *Simulated Annealing*

SIMD *Single Instruction Multiple Data*

SISD *Single Instruction Single Data*

SKEPU *Skeleton Programming Framework for Multicore CPU*

SKETO *SKEletons in TOkyo*

SPEC *Standard Performance Evaluation Corporation*

SPLASH *Stanford ParalleL Applications for Shared-Memory*

SPH *Smoothed Particle Hydrodynamics*

SRAD *Speckle Reducing Anisotropic Diffusion*

STAP *Space-Time Adaptive Processing*

STL *Standard Template Library*

TBB *Threading Building Blocks*

TFLOPS *Tera Floating-Point Operations Per Second*

UE *Unión Europea*

VASARI *Visual Arts System for Archiving and Retrieval of Images*

Bibliografía

[1] J. Dempsey, "Fluid Animate Particle Simulation." *Dr. Dobb's, The world of software development*, Marzo 2010. [En línea]. Disponible en: http://www.drdobbs.com/parallel/fluid-animate-particle-simulation/228800371.

[2] B. Staff, "What Is Ray Tracing and How It Improves Graphics in Video Games?." *Beebom*, Marzo 2018. [En línea]. Disponible en: https://beebom.com/what-is-ray-tracing-how-improves-graphics-video-games/.

[3] M. Wright and al., "The opportunities and challenges of exascale computing," *U.S. Department of Energy*, 2010. [En línea] Disponible en: https://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf.

[4] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Septiembre 1972. [En línea]. Disponible en: https://pdfs.semanticscholar.org/5af1/24cf8cb77d89513f61b5c29ac60a738521e6.pdf.

[5] "ISO/IEC 14882:2011 - Information technology – Programming languages – C++." *ISO/IEC Standard*, Geneva, Switzerland: International Organization for Standardization, Septiembre 2011.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, pp. 40–53, Marzo 2008. [En línea]. Disponible en: http://delivery.acm.org/10.1145/1370000/1365500/p40-nickolls.pdf?ip=83.46.108.191&id=1365500&acc=OPEN&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E6D218144511F3437&__acm__=1529167285_d47c580e67c618355837f155a9453124.

[7] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, pp. 66–73, Mayo 2010.

[8] A. Guillon, "An Introduction to OpenCL C++," *The Khronos Group Inc*, 2015. [En línea]. Disponible en: https://www.khronos.org/assets/uploads/developers/resources/Intro-to-OpenCL-C++-Whitepaper-May15.pdf.

[9] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, pp. 46–55, Enero 1998.

[10] J. Reinders, "Outfitting C++ for Multi-Core Processor Parallelism," *Intel Threading Building Blocks*, Julio 2007. [En línea]. Disponible en: http://index-of.es/Programming/C++/O'Reilly%20Intel%20Threading%20Building%20Blocks%20OutFitting%20C++%20for%20Multi-Core%20Processor%20Parallelism.pdf.

[11] "ISO/IEC 14882:2017 - Information technology – Programming languages – C++." *ISO/IEC Standard*. Geneva, Switzerland: International Organization of Standarization, Diciembre 2017.

[12] H. A. Gabb, "Transform Sequential C++ code to Parallel with Parallel Stl," *Intel Parallel Universe Magazine*, Abril 2017. [En línea]. Disponible en: https://sos-software.com/wp-content/uploads/Intel-Parallel-Universe-Magazin-Ausgabe-28.pdf.

[13] G. Horacio and L. Mario, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software: Practice and Experience*, vol. 40, pp. 1135–1160, Noviembre 2010.

[14] M. Vanneschi, "The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications," *Parallel Computing*, vol. 28, pp. 1709–1732, Diciembre 2002.

[15] P. Ciechanowicz, M. Poldner, and H. Kuchen, "The münster skeleton library muesli: A comprehensive overview," *ERCIS Working Papers*, 2009. [En línea]. Disponible en: `https://www.ercis.org/sites/ercis/files/structure/network/research/ercis-working-papers/ercis_wp_07.pdf`.

[16] J. Enmyren and C. W. Kessler, "Skepu: A multi-backend skeleton programming library for multi-gpu systems," in *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, (New York, NY, USA), pp. 5–14, ACM, 2010.

[17] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming," in *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, (New York, NY, USA), ACM, 2006.

[18] M. Torquati, M. Aldinucci, and M. Danelutto, "Fastflow (ff)," Marzo 2015. [En línea]. Disponible en: `http://calvados.di.unipi.it/`.

[19] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Targeting distributed systems in fastflow," in *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing*, vol. 7640 of *LNCS*, pp. 47–56, Springer, 2013.

[20] M. Torquati, "Parallel programming using fastflow," *Computer Science Department, University of Pisa*, Septiembre 2015. [En línea]. Disponible en: `http://calvados.di.unipi.it/storage/tutorial/fftutorial.pdf`.

[21] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García, "A generic parallel pattern interface for stream and data processing," *Concurrency and Computation: Practice and Experience*, vol. 29, p. e4175, Abril 2017. [En línea]. Disponible en: `https://onlinelibrary.wiley.com/doi/epdf/10.1002/cpe.4175`.

[22] "Published SPEC Benchmark Results." *SPEC Benchmark: Standard Performance Evaluation Corporation*, Diciembre 2017. [En línea]. Disponible en: `http://www.spec.org/results.html`.

[23] J. Hardman, "NAS Parallel Benchmarks." *NASA Advanced Supercomputing Division*, Marzo 2016. [En línea]. Disponible en: `https://www.nas.nasa.gov/publications/npb.html`.

[24] K. C. Cain, J. A. Torres, and R. T. Williams, "RT_STAP: Real-Time Space-Time Adaptive Processing Benchmark," *MITRE CORP BEDFORD MA*, Octubre 1997. [En línea]. Disponible en: `http://www.dtic.mil/dtic/tr/fulltext/u2/a340497.pdf`.

[25] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 101–111, Abril 2016. [En línea]. Disponible en: `http://ditec.um.es/~aros/papers/pdfs/csakalis-ispass16.pdf`.

[26] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, Octubre 2009. [En línea]. Disponible en: `http://www.cs.virginia.edu/~skadron/Papers/rodinia_iiswc09.pdf`.

[27] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, Octubre 2008. [En línea]. Disponible en: `http://parsec.cs.princeton.edu/doc/parsec-report.pdf`.

[28] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, Junio 2009. [En línea]. Disponible en: http://parsec.cs.princeton.edu/publications/bienia09parsec-2.0.pdf.

[29] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto, "Bringing parallel patterns out of the corner: The p3 arsec benchmark suite," *ACM Trans. Archit. Code Optim.*, vol. 14, pp. 33:1–33:26, Octubre 2017. [En línea]. Disponible en: http://pages.di.unipi.it/mencagli/publications/preprint-taco-2017.pdf.

[30] The Institute of Electrical and Electronics Engineers, Inc., "IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1998*, pp. 1–40, Octubre 1998. [En línea]. Disponible en: http://www.math.uaa.alaska.edu/~afkjm/cs401/IEEE830.pdf.

[31] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García, "GrPPI: Generic Reusable Parallel Pattern Interface." GitHub, 2018. [En línea]. Disponible en: https://github.com/arcosuc3m/grppi.

[32] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, NJ, USA, Enero 2011. [En línea]. Disponible en: http://parsec.cs.princeton.edu/publications/bienia11benchmarking.pdf.

[33] J. Hull, *Options, futures, and other derivatives, eighth edition, global edition.* Pearson Education, 2012. [En línea]. Disponible en: https://shamit8.files.wordpress.com/2014/11/options-futures-and-other-derivatives-8th-john.pdf.

[34] "POSIX Threading Library." *GNU Hurd*, Febrero 2015. [En línea]. Disponible en: https://www.gnu.org/software/hurd/libpthread.html.

[35] "GNU Free Documentation License 1.2." *GNU Operating System*, Noviembre 2016. [En línea]. Disponible en: https://www.gnu.org/licenses/old-licenses/fdl-1.2.html.

[36] "OpenMP Application Programming Interface," *Open Multi-Processing*, Noviembre 2015. [En línea]. Disponible en: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[37] "How is Intel® TBB licensed?." Intel® Threading Building Blocks (Intel® TBB), 2017. [En línea]. Disponible en: https://www.threadingbuildingblocks.org/faq/10.

[38] "Apache license version 2.0." *The Apache Software Foundation!*, Enero 2004. [En línea]. Disponible en: https://www.apache.org/licenses/LICENSE-2.0.

[39] "Downloads and contacts." FastFlow, Marzo 2018. [En línea]. Disponible en: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:downloads.

[40] "GNU Lesser General Public License." *GNU Operating System*, Noviembre 2016. [En línea]. Disponible en: https://www.gnu.org/licenses/lgpl-3.0.en.html.

[41] "Licencias." *El sistema operativo GNU*, Abril 2018. [En línea]. Disponible en: https://www.gnu.org/licenses/licenses.es.html.

[42] "Licensing of distributions." *The Apache Software Foundation!* [En línea]. Disponible en: http://www.apache.org/licenses/.

[43] "List of Licenses." *The PARSEC Benchmark Suite License: Princeton University*, 2009. [En línea]. Disponible en: http://parsec.cs.princeton.edu/license.htm.

[44] "GNU General Public License." *GNU Operating System*, Noviembre 2016. [En línea]. Disponible en: https://www.gnu.org/licenses/gpl-3.0.en.html.

[45] "GNU General Public License, version 2." *GNU Operating System*, Septiembre 2017. [En línea]. Disponible en: https://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

[46] "License / Copyright Information." *The Mesa 3D Graphics Library*. [En línea]. Disponible en: https://www.mesa3d.org/license.html.

[47] "Protección de datos de carácter personal." *Agencia Estatal Boletín Oficial del Estado*, Mayo 2018. [En línea]. Disponible en: https://www.boe.es/legislacion/codigos/codigo.php?id=055_Proteccion_de_Datos_de_Caracter_Personal&modo=1.

[48] "ISO/IEC 14882:2014 - Information technology – Programming languages – C++." *ISO/IEC Standard*, Geneva, Switzerland: International Organization for Standardization, Diciembre 2014.

[49] "Bases y tipos de cotización 2018." *Ministerio de Empleo y Seguridad Social*, 2018. [En línea]. Disponible en: http://www.seg-social.es/Internet_1/Trabajadores/CotizacionRecaudaci10777/Basesytiposdecotiza36537/index.htm.

[50] "Procesador Intel® Xeon® E5-2695 v2." *Intel® ARK*. [En línea]. Disponible en: https://ark.intel.com/es-es/products/75281/Intel-Xeon-Processor-E5-2695-v2-30M-Cache-2_40-GHz.

[51] P. Lathan, "RAM Price Report: DDR4 Same Price as Initial Launch." *Gamers Nexus*, Enero 2018. [En línea]. Disponible en: https://www.gamersnexus.net/industry/3212-ram-price-investigation-ddr4-same-price-as-initial-launch.

[52] "Average selling price of personal computers (pcs) worldwide from 2015 to 2019, in actual and constant currency (in u.s. dollars)." *The Statistics Portal*, Julio 2017. [En línea]. Disponible en: https://www.statista.com/statistics/722992/worldwide-personal-computers-average-selling-price/.